IBM External Submission #24167

Reprint Courtesy of International Business Machines Corporation, © 2015 International Business Machines Corporation

INTERNATIONAL BUSINESS MACHINES CORPORATION (IBM) ARMONK, NEW YORK 10504

PERMISSION TO REPRINT IBM COPYRIGHTED PUBLICATIONS

Each reprint must be accompanied by the following credit line: **"Reprint Courtesy of International Business Machines Corporation, © (year) International Business Machines Corporation".** The credit line normally should appear on the page where the reprint appears, either under the title or as a footnote.

If the foregoing is inconvenient, the credit line may be placed on the face or back of the title page (or front cover, if there is no title page) or in a conveniently viewable manner with suitable reference to the place where the reprint appears.

When multiple IBM materials are reprinted in the same publication, a consolidated credit paragraph may be used on the title page, or in a conveniently viewable manner listing the titles, corresponding copyright notices and references to the points where the reprints appear.

It is the understanding of **International Business Machines Corporation** that the purpose for which its material is being used is accurate and true as stated in the original request.

Permission to quote from, transmit electronically or reprint IBM material is limited to the purpose and quantities originally requested and must not be construed as a blanket license to use the material for other purposes or to reprint other IBM copyrighted material.

IBM reserves the right to withdraw permission to use copyrighted material whenever, in its discretion, it feels that the privilege of using its material is being used in a way detrimental to its interest or the above instructions are not being followed properly to protect its copyright.

No permission is granted to use trademarks of **International Business Machines Corporation** and its affiliates apart from the incidental appearance of such trademarks in the titles, text, and illustrations of the named publications. Any proposed use of trademarks apart from such incidental appearance requires separate approval in writing and ordinarily cannot be given. The use of any IBM trademark should not be of a manner which might cause confusion of origin or appear to endorse non-IBM products.

THIS PERMISSION IS PROVIDED WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

INTERNATIONAL BUSINESS MACHINES CORPORATION

Dated: August 19, 2015

Workplace Shell Programming Guide

IBM OS/2 Toolkit Documentation

Introduction

This book is a programmer's guide to the Workplace Shell*, an application environment integrated with the Operating System/2* (OS/2*), with an object-oriented user interface using IBM*'s System Object Model (SOM).

The Workplace Shell: Object-Oriented User/Programming Interface

The Workplace Shell constitutes the interface with which users interact with the operating system. It represents items that users most frequently manipulate as familiar objects on a Desktop. As a user interface, the Workplace Shell conforms to many of the guidelines described by the Common User Access* (CUA*).

In addition to being an interface through which a user can organize and work with the operating system, the Workplace Shell provides its programming interface to you. This means that applications can blend into the operating system's user interaction model, allowing users to work with new applications in the same manner as the operating system itself.

By exploiting the Workplace Shell's application programming interface, you can also reuse and build on much of the functionality already implemented by the Workplace Shell. By using the Workplace Shell methods, an application can implement its interface so that it includes familiar Workplace Shell user interaction tools such as:

- Objects presented to the user on the Desktop
- Common mouse/keyboard interaction techniques
- Settings notebook to manipulate attributes of the application
- Dynamic context menus to present operational choices to user
- Object mobility via Drag and Drop operation
- Application view management with the OS/2 Window list.

In addition to the portion of the functions devoted to managing the presentation of the application's user interface, the Workplace Shell also provides functions (methods) to help coordinate the implementation of the application.

Objectives of this Guide

This guide provides information on topics relevant to writing application programs for the Workplace Shell. These topics include:

- Creating and Installing Workplace Shell Objects
- Debugging Workplace Shell Applications
- Object Aid: Help Methods
- Object Containers: Container Methods
- Object Criteria: Details Methods
- Object Errors: Error Handling Methods
- Object Information: Set/Query Methods
- Object Initialization and Termination: Setup/Cleanup Methods
- Object Memory: Memory Allocation Methods
- Object Mobility: Direct Manipulation Methods
- Object Persistence: Save/Restore Methods
- Object Properties: Notebook Methods
- Object Usage: Usage Methods

- Object User Action: Pop-Up Menu Methods
- REXX Utility Workplace Shell Functions
- Workplace Shell and System Object Model
- Workplace Shell Distributed SOM
- Workplace Shell Processes and Threads
- Workplace Shell Programming Interface
- Workplace Shell Win Functions

The contents of each of these topics is briefly described in the following sections.

Creating and Installing Workplace Shell Objects

This chapter walks you through the Car sample which is included in the Toolkit, and through WPStyler, a complete sample program. The purpose is to provide detail information on how to create an object. For each sample, the different Workplace Shell instance and class methods used, along with the new methods introduced by each of them, are explained. Once objects are created, they can be installed on the Desktop in two ways: by running an installation program or batch file, or by using the Workplace Shell Class List Object Utility. As objects can also reside on the Toolbar, this chapter describes the different setup strings and methods used with the Toolbar. Finally, this chapter explains how to support printing the contents of an object from the Desktop.

Debugging Workplace Shell Applications

Debugging a Workplace Shell application is somewhat like debugging a Presentation Manager* (PM*) application. But because Workplace Shell objects are implemented via DLLs instead of EXEs, they can be more of a challenge. This chapter describes the SOM and Workplace Shell facilities and techniques to aid debugging applications.

Object Aid: Help Methods

While running an application or viewing a document, the user may require help on the application or document. It is important to include the help information for all applications as it is used often. This chapter describes different methods that can be overridden in order to provide help from an application window, when a user requests it.

Object Containers: Container Methods

A *container* is a visual component that holds objects. CUA container control is the primary way of interacting with Workplace Shell objects. All objects with which the user interacts are simply records that have been inserted into a container control. Any Workplace Shell object can be inserted into any container control created on the Workplace process. This chapter describes the different views available for containers, and the methods used to create and manipulate containers in Workplace Shell applications.

Object Criteria: Details Methods

The class defines the object's instance data and methods. *Details data* are part of instance data and define more accurately the behavior of the object. This chapter describes the general concepts of Details data and explains how to use the Details methods in Workplace Shell applications.

Object Errors: Error Handling Methods

The Workplace Shell provides methods that you can override to design your own error handling system, using error objects. This chapter provides information on error handling methods for Workplace Shell applications.

Object Information: Set/Query Methods

The Workplace Shell provides several set and query methods that you can use to set or to get object information, such as the default view of the object's in-use list, the object's icon, and the object's title. This chapter provides information for using object information methods. It describes how to define the behavior of Workplace Shell objects, using applicable styles.

Object Initialization and Termination: Setup/Cleanup Methods

The Workplace Shell provides methods that you can use to setup the characteristics and behaviors of an object. Likewise, it also provides methods that you can use to cleanup after an object is no longer in use. This chapter provides information on setup and cleanup methods that are available in the Workplace Shell.

Object Memory: Memory Allocation Methods

The Workplace Shell provides methods which you can override to design your own memory management system, or utilize to manage memory required for objects of your application. This chapter provides information on memory management methods for Workplace Shell applications.

Object Mobility: Direct Manipulation Methods

Direct manipulation refers to the OS/2 object mobility. That is, it refers to the OS/2 user capability of moving an object from one container to another by dragging an object icon from a source container and dropping it on a target container. Direct manipulation methods allow you to specify how objects that are directly effected by such a move communicate with each other. This chapter outlines the protocols of direct manipulation with respect to their interactions with the Workplace Shell.

Object Persistence: Save/Restore Methods

The WPObject class provides methods that support persistent objects, that is, objects for which the system saves information. These methods include save and corresponding restore methods. This chapter provides information using these state methods in Workplace Shell applications.

Object Properties: Notebook Methods

A notebook is a visual component used to display the setttings for an object and to enable the user to change them. This chapter describes the General and Window pages of the Settings notebook, and explains how to use the notebook methods in Workplace Shell applications.

Object Usage: Usage Methods

Every Workplace Shell object in the system has an in-use list. Object usage methods allow an object to keep track of its resources and how it is being used. This chapter provides information for using in-use lists that are associated with objects.

Object User Action: Pop-Up Menu Methods

This chapter describes the pop-up context menus that can be used to invoke actions on Workplace Shell objects. It also provides an overview of how to create and manipulate the pop-up context menu for your Workplace Shell objects.

REXX Utility Workplace Shell Functions

REXX combines the simplicity of a programming language such as BASIC with features that exist in more powerful languages such as C, PASCAL, or PL/1. Part of the success of REXX has been due in part to having been selected as the Systems Application Architecture* Programming Language (SAA/PL). The purpose is to provide a common look and feel across all operating systems. This is important because CUA is the standard that the Workplace Shell uses which is also part of SAA* REXX provides the facility for the Workplace Shell to create, modify, and delete objects. This chapter describes the REXX functions and how to use them in Workplace Shell applications.

Workplace Shell and System Object Model

SOM is the object technology used for the implementation of the Workplace Shell. It provides a language neutral, object-oriented programming methodology that you can use to create Workplace Shell objects and bind them to the Workplace Shell. This chapter describes SOM's architecture, its features and environment. It describes its methods, classes, and finally provides an example of its workings. This chapter is not intended to be a SOM programming guide. It is, however, intended to serve as an introduction to SOM at a level necessary to write Workplace Shell applications.

Workplace Shell Distributed SOM

Distributed SOM (DSOM) provides a framework that allows application programs to access objects across address spaces. That is, application programs can access objects in other processes, even on different machines. Both the location and implementation of an object are hidden from the client which accesses the object (using method calls) in the same manner regardless of its location.

Workplace Shell Processes and Threads

The Workplace Shell's current implementation exploits the multi-threading capabilities of the OS/2 operating system. This chapter describes how the Workplace Shell is initialized, the relationship between its various threads, and ramifications on Workplace Shell method invocation.

Workplace Shell Programming Interface

The Desktop (which also is an object), the objects that appear on the Desktop, and the underlying code supporting these objects constitute

the OS/2 Workplace Shell, the default OS/2 user interface. Likewise, methods in the Workplace Shell are methods that were implemented for the creation of the Desktop object and its objects. This chapter describes Workplace Shell classes and characteristic behavior of Workplace Shell objects.

Workplace Shell Win Functions

Applications cannot call Workplace Shell objects' methods directly. They are not clients of Workplace Shell objects, in the same sense that applications can be clients of SOM objects. Workplace Shell objects are derived from the WPObject class, which, in turn, is derived from the SOMObject class. They share all the features of SOM objects but only the Workplace Shell can directly manipulate them. Because there are times when applications might need to effect changes to the Desktop and objects on the Desktop, the Workplace Shell provides functions that permit you to proceed these changes. This chapter describes the Workplace Shell Win Functions.

Creating and Installing Workplace Shell Objects

This chapter walks you through the CAR sample which is included in the Toolkit, and through WPStyler, a complete sample program. The purpose is to provide detail information on how to create an object. For each sample, the different Workplace Shell instance and class methods used, along with the new methods introduced by each of them, are explained. Once objects are created, they can be installed on the Desktop in two ways: by running an installation program or batch file, or by using the Workplace Shell Class List Object Utility. As objects can also reside on the Toolbar, this chapter describes the different setup strings and methods used with the Toolbar. Finally, this chapter explains how to support printing the contents of an object from the Desktop.

About Workplace Shell Objects

This section describes the:

- CAR sample and its implementation, which consists of:
 - Providing for persistence of the object
 - Changing Settings notebook pages
 - Removing an Item from the pop-up menu
 - Adding pop-up menu items
 - Creating and registering a new open view
 - Processing new pop-up menu items
 - Processing help for new pop-up menu items
- Creation of new instances of objects
- Printing objects
- Toolbar.

Creation of Objects

The CAR sample Workplace Shell object included in the Toolkit provides an example of how to create an object. CAR is a spin-off implementation of the CAR object example used in the *Systems Application Architecture: Common User Access Guide to User Interface Design* .

The CAR object has two views:

- An Open car view, which is a representation of a CAR that moves randomly around a window.
- A Settings notebook that permits the user to change the sound of the horn, on the Horn beep page, and the speed of the CAR, on the Dashboard page.

The Horn beep and Dashboard pages in the CAR's Settings notebook are dialog windows whose contents are defined in a dialog template in the CAR.RC resource file. The resources are appended to the binary CAR.DLL file. The help panels associated with the dialogs are defined in the CAR.HLP help library. The association between the help panels and the dialogs is established with the dialog template in the resource file.

CAR is a persistent object but not a part of the file system. Therefore, it is derived from the WPAbstract class, which is derived from the WPObject root Workplace Shell class. This means that CAR inherits the methods from WPAbstract, which, in turn, inherits all of the instance and class methods from WPObject. CAR defines new methods and overrides some instance and class methods inherited from both WPAbstract and WPObject. New methods defined for the CAR class are summarized in the following table:

Method Name	Description
carQueryInfo	Gets the CAR information. It is called by the dialog procedure to get the latest CAR data when the dialog window opened.
carSetInfo	Sets up the CAR information. It is called by the dialog procedure to update the CAR data as the user interacts with the dialog window.
wpAddDashboardPage	Inserts the Dashboard page into the Settings notebook. It is called by wpAddSettingsPages.
wpAddHornBeepPage	Inserts the Horn beep page into the Settings notebook. It is called by wpAddSettingsPages.

Implementation of the CAR Object

The implementation of the CAR object consists primarily of overrides to instance and class methods inherited from the WPObject and WPAbstract classes. The following sections describe the different instance and class methods that are overridden.

Providing for Persistence of the CAR Object

Because the CAR class defines its own set of data, it overrides wpSaveState and wpRestoreState. These method overrides, in turn, call wpSaveLong and wpRestoreLong to save instance data defined for the CAR class.

Changing Settings Notebook Pages

CAR creates Settings notebook pages by overriding wpAddSettingsPages inherited from WPObject. The wpAddSettingsPages method calls the new methods defined by the CAR class, wpAddDashboardPage and wpAddHornBeepPage. In turn, each of these methods calls wpInsertSettingsPage inherited from WPObject.

Removing an Item from the Pop-Up Menu

Because the user item **Print a car** does not make sense for this object, the CAR class removes the **Print** item from the CAR object's pop-up menu by overriding wpFilterPopupMenu inherited from WPOject.

Adding Pop-Up Menu Items

The CAR class adds the **Beep horn** item to its primary pop-up menu by overriding wpModifyPopupMenu inherited from WPObject. The method override for wpModifyPopupMenu calls wpInsertPopupMenuItems, also inherited from WPObject, to insert the item into the CAR's pop-up menu.

The CAR class also adds the **Open car** item (OPEN_CAR) to its **Open** submenu by overriding wpModifyPopupMenu. The method override calls wpInsertPopupMenuItems to insert **Open car** into the **Open** submenu.

Creating and Registering a New Open View

A window for the OPEN_CAR view is created and opened by calling WinCreateStdWindow. The window procedure for the client of the OPEN_CAR window registers the OPEN_CAR view and associates the OPEN_CAR view with that window by calling wpRegisterView.

Processing New Pop-Up Menu Items

When a class defines new items for its menus, it must provide for the processing of these items when the user selects an item. This is done by overriding wpMenuItemSelected inherited from WPObject. Depending on the item the user selects, the method override for wpMenuItemSelected calls the object's wpOpen to open the OPEN_CAR view or calls carBeepHorn to beep the CAR's horn.

Processing Help for New Pop-Up Menu Items

When a class defines new items for its pop-up menu, it also must provide for the processing of the help for these items when the user requests it. This is accomplished by overriding wpMenuItemHelpSelected inherited from WPObject. Given the item selected by the user, the method override for wpMenuItemHelpSelected calls wpDisplayHelp inherited from WPObject to display the Help dialog window for that item. The wpDisplayHelp method requires the ID for the help panel associated with the item, as well as the name of the help library where it resides.

The implementation of the CAR object also demonstrates the use of:

- A release order list in CAR.IDL.
- External stem and prefix attributes in the class section of CAR.IDL. This allows the debugging process to be easier, because method names are externalized.
- A message queue for the Workplace Shell. Object code runs on the Workplace Shell's thread. Windows associated with objects receive messages through the Workplace Shell's message queue. Objects do not need their own message queues.

Creation of New Instances of Objects

An object template is the primary user mechanism for creating new instances of objects. Specifically, a template is a state of an object where the default Drag operation is **Create another**, that is, dragging and dropping the template results in the creation of an instance of the object. The visual representation of a template is the object's icon on top of a "yellow sticky pad" with the top sheet slightly peeled up, as shown in the following figure:



Any object that supports the **Create another** item can be changed by the user to and from a template state by selecting the template checkbox on the General page in the object's Settings notebook. A template is created automatically when a class is registered, unless the class wpclsQueryStyle returns CLSSTYLE_NEVERTEMPLATE.

When the operating system is first installed, template objects reside in the templates folder on the desktop. The templates folder always contains a template object for each class of object installed on the system that supports the **Create another** item. Any new object registered by WinRegisterObjectClass that supports the **Create another** item automatically appears in this folder. A template for each object class registered using this function cannot be removed from the templates folder.

Printing Objects

To support printing the contents of an object from the Desktop-by using the object's pop-up menus or dragging the object and dropping it on the printer object-the object's class definition must override wpPrintObject inherited from its parent class.

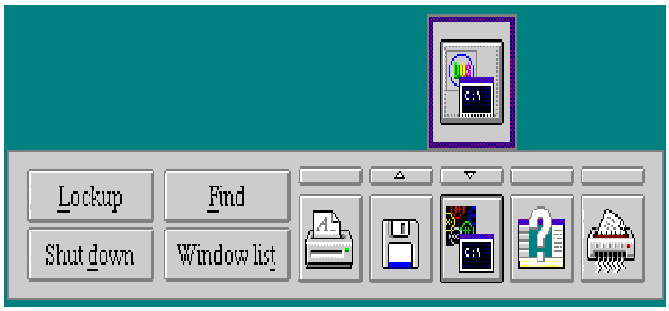
The new class is created as a subclass of an existing class. For example, if the object is to be a data file, the WPDataFile class would be used as the parent class. In the class definition file, the new class overrides wpPrintObject that it inherits from its parent class. The new class' version of wpPrintObject contains the code that prints the contents of the object.

Note: It is recommended that the code in the object's version of wpPrintObject should start on a separate thread to execute the code that actually prints the contents of the object. By doing this, the control can be returned to the Workplace Shell and the user immediately. Supporting code, such as dialog windows, should be done on a separate thread as well.

Your new class should allow data files that do not belong to your class to be printed as well. To do this, you should also override wpclsQueryInstanceType and wpclsQueryInstanceFilter to verify if a file is yours (that is, if its .TYPE extended attribute denotes your new class), or if it is not yours. If the file is not yours, you will want to call parent_wpPrintObject to do the printing.

About Toolbars

The Toolbar is a convenient area for users. It provides them with a place to get fast access to their frequently used objects and Desktop actions. In addition to placing objects on the Toolbar, users may place objects in drawers. A drawer is represented by the little button above an object on the Toolbar, as shown in the following figure:



The button of an open drawer contains a downward pointing arrow. The button of a closed drawer contains a upward pointing arrow. As shown in the above figure, the drawer above the OS/2 Window object is open and contains the DOS Window object.

While there is no restriction on the content of the drawer, its most logical function is to hold objects that are either related or similar to the object on the Toolbar but are less frequently used.

There is no limit to the number of Toolbars that may exist in the system. The system Toolbar is defined as the one with an OBJECTID of <WP_LAUNCHPAD>. When the user double clicks the select button on a folder (or Desktop background space) the Workplace Shell looks for the system Toolbar. For example, you can configure a different Toolbar for a work area, so that each time that work area is opened, its Toolbar is opened.

Toolbar Setup Strings

The Toolbar is highly customizable. You can use the following setup strings to customize the Toolbar:

Keyname	Value	Description
DRAWEROBJECTS	A drawer number followed by a comma delimited set of object IDs or path and file names. Separate the drawer number and the first object with a comma.	Adds the objects to the end of the numbered Toolbar drawer: 0=Toolbar, 1=left-most drawer, etc.
FPOBJECTS	A comma-delimited set of object IDs or path and file names.	Adds the objects to the end of the Toolbar.
LPACTIONSTYLE	MINI	Displays the action buttons as mini-icons.
	NORMAL	Displays the action buttons as normal or large icons.
	OFF	Turns off the display of action buttons.
	TEXT	Displays the actions buttons as text. This is the default.
LPCLOSEDRAWER	YES	Closes the Toolbar drawers after opening an object.
	NO	Leaves the Toolbar drawers open after opening an object.
LPDRAWERTEXT	YES	Displays titles of objects located in drawers. This has no effect on objects located on the Toolbar.
	NO	Hides titles of objects located in drawers. This has no effect on objects located on the Toolbar.
LPFLOAT	YES	Keeps the Toolbar floating on top of all other windows.
	NO	Does not keep the Toolbar floating on top of all other windows.
LPHIDECTLS	YES	Hides the frame controls (title bar and system menu). This is the default.
	NO	Displays the frame controls (title bar and system menu).
LPSMALLICONS	YES	Displays objects using small icons.
	NO	Display objects using large icons.
LPTEXT	YES	Displays titles of

		objects located on the Toolbar. This has no effect on objects located in drawers.
	NO	Hides titles of objects located on the Toolbar. This has no effect on objects located in drawers.
LPVERTICAL	YES	Displays the Toolbar vertically.
	NO	Displays the Toolbar horizontally. This is the default.

These keynames can be used with the following Win and REXX functions:

- WinCreateObject
- WinSetObjectData
- SysCreateObject
- SysSetObjectData.

The following example adds two objects to the first drawer of the Toolbar:

```
SysSetObjectData("<WP_LAUNCHPAD>",
                 "DRAWEROBJECTS=1,C:\README.TXT,<WP_MINDEX>")
```

The following example causes the Toolbar to float on top:

```
SysSetObjectData("<WP_LAUNCHPAD>", "LPFLOAT=YES")
```

Toolbar Set Methods

In addition to setup strings, methods shown in the following table provide the same functionality:

Method Name	Value	Description
wpSetActionButtonStyle	ACTION_BUTTONS_MINI	Displays the actions buttons as mini-icons.
	ACTION_BUTTONS_NORMAL	Displays the action buttons as normal or large icons.
	ACTION_BUTTONS_OFF	Turns off the display of action buttons.
	ACTION_BUTTONS_TEXT	Displays the action buttons as text. This is the default.
wpSetCloseDrawer	TRUE	Closes the Toolbar drawer after opening an object.
	FALSE	Leaves the Toolbar drawer open after opening an object.
wpSetDisplaySmallIcons	TRUE	Displays objects using small icons.

	FALSE	Displays objects using large icons.
wpSetDisplayText	TRUE	Displays titles of objects located on the Toolbar. This has no effect on objects located in drawers.
	FALSE	Hides titles of objects located on the Toolbar. This has no effect on objects located in drawers.
wpSetDisplayTextInDrawers	TRUE	Displays titles of objects located in drawers. This has no effect on objects located on the Toolbar.
	FALSE	Hides titles of objects located in drawers. This has no effect on objects located on the Toolbar.
wpSetDisplayVertical	TRUE	Displays the Toolbar vertically.
	FALSE	Displays the Toolbar horizontally. This is the default.
wpSetFloatOnTop	TRUE	Keeps the Toolbar floating on top of all other windows.
	FALSE	Does not keep the Toolbar floating on top of all other windows.
wpSetHideLaunchPadFrameCtls	TRUE	Hides the frame controls (title bar and system menu). This is the default.
	FALSE	Displays the frame controls (title bar and system menu).
wpSetObjectListFromHObjects	A drawer number.	Adds the objects to the numbered drawer of the Toolbar.
	The number of objects to be added.	
	An array of HOBJECTS.	
	A position within the drawer.	0=Toolbar, 1=left-most drawer, and so on. Use ADD_OBJECT_FIRST to add the objects at the beginning of the drawer. Use ADD_OBJECT_LAST to add the objects at the end of the drawer. Use a position number to state which objects the new objects should be inserted after (0=first object).

wpSetObjectListFromObjects	A drawer number.	Adds the objects to the specified Toolbar drawer.
	The number of objects to be added.	
	An array of WPObject*.	
	A position with the drawer.	0=Toolbar, 1=left-most drawer, and so on. Use ADD_OBJECT_FIRST to add the objects at the beginning of the drawer. Use ADD_OBJECT_LAST to add the objects at the end of the drawer. Use a position number to state which objects the new objects should be inserted after (0=first object).
wpSetObjectListFromStrings	A drawer number.	Adds the objects to the specified Toolbar drawer.
	A NULL delimited set of object IDs or path and file names terminated by a double NULL.	
	A position with the drawer.	0=Toolbar, 1=left-most drawer, and so on. Use ADD_OBJECT_FIRST to add the objects at the beginning of the drawer. Use ADD_OBJECT_LAST to add the objects at the end of the drawer. Use a position number to state which objects the new objects should be inserted after (0=first object).

The following example adds two objects to the first drawer in the Toolbar:

```
PSZ pszObjectList = "C:\\README.TXT\0<WP_MINDEX>\0";
_wpSetObjectListFromStrings(somSelf,
                            1,
                            pszObjectList,
                            ADD_OBJECT_LAST);
```

The following example causes the Toolbar to float on top:

```
_wpSetFloatOnTop(somSelf,TRUE);
```

Toolbar Query Methods

The following table provides the methods for querying information about the state of the Toolbar:

Method Name	Description
wpQueryActionButtons	Returns a pointer of array of ACTION data structures. Each ACTION structure has the following elements: o Pointer to the text to be displayed o Handle to the icon to be displayed o Menu item number that is passed to the Desktop via wpMenuItemSelected, when the action button is pressed. The number of ACTION structures in the array is returned using parameters.
wpQueryActionButtonStyle	Returns the current setting for the action button style: o ACTION_BUTTONS_MINI o ACTION_BUTTONS_NORMAL o ACTION_BUTTONS_OFF o ACTION_BUTTONS_TEXT.
wpQueryCloseDrawer	Returns TRUE if the drawer is closed after one of its objects is opened. Otherwise, it returns FALSE.
wpQueryDisplaySmallIcons	Returns TRUE if objects' icons are displayed using small or mini-icons. Otherwise, it returns FALSE.
wpQueryDisplayText	Returns TRUE if titles of objects located on the Toolbar are displayed. Otherwise, it returns FALSE.
wpQueryDisplayTextInDrawers	Returns TRUE if text is displayed for objects that are located in drawers. Otherwise, it returns FALSE.
wpQueryDisplayVertical	Returns TRUE if the Toolbar is displayed vertically. Otherwise, it returns FALSE.
wpQueryFloatOnTop	Returns TRUE if the Toolbar is set to float on top of all other windows. Otherwise, it returns FALSE.
wpQueryHideLaunchPadFrameCtls	Returns TRUE if the frame controls (title bar and system menu) are hidden. Otherwise, it returns FALSE.
wpQueryObjectList	Returns a pointer to an array of HOBJECT that represents the list of objects in the drawer. The number of objects is returned using parameters.

Using Workplace Shell Objects

This section describes how to:

- Set objects on the Toolbar
- Create objects
- Install objects
 - By running an installation program
 - By running a batch file
 - By using the Workplace Shell Class List Object Utility.

Using Toolbars

Action buttons are added to or removed from the Toolbar by overriding wpQueryActionButtons. You must declare an instance data variable in your subclass of type PACTIONS. When overriding wpQueryActionButtons, verify if pMyActions is NULL. If it is, allocate memory to hold the parent's actions and the ones you are adding. Copy the parent's actions to your array, add yours, and then return, as shown in the following sample code:

```
SOM_Scope PACTIONS SOMLINK mylnchpd_wpQueryActionButtons(
                           WPLaunchPad *somSelf,
                           PULONG pulNumActions)
{
  MyLaunchPadData *somThis = MyLaunchPadGetData(somSelf);
  PACTIONS pParentActions;
  ULONG ulParentActions;

  MyLaunchPadMethodDebug("MyLaunchPad","mylnchpd_wpQueryActionButtons");

  if (!_pMyActions)
  {
    pParentActions = parent_wpQueryActionButtons(somSelf,
                                                 &ulParentActions);
    _ulMyActions   = ulParentActions + 1;
    _pMyActions    = _wpAllocMem(somSelf,_ulMyActions*sizeof(ACTIONS), NULL);

    if (!_pMyActions)
    {
      return parent_wpQueryActionButtons(somSelf, pulNumActions);
    }
    _pMyActions[ulParentActions].pszTitle = "My new action";
    _pMyActions[ulParentActions].ulMenuId = MY_MENUID;
    _pMyActions[ulParentActions].hIcon    = myActionIcon;

    if (ulParentActions)
    {
      memcpy(_pMyActions, pParentActions,ulParentActions*sizeof(ACTIONS));
    }
  }

  if (pulNumActions)
  {
    (*pulNumActions) = _ulMyActions;
  }
  return (PACTIONS) _pMyActions;
}
```

Note: When overriding wpQueryActionButtons you must also provide a subclass of WPDesktop, override wpMenuItemSelected, and handle the MY_MENUID action.

Creating Objects

In order to illustrate what is required to create a complete Workplace Shell object that you can use constructively, a complete Workplace Shell SOM program is included in Sample Code for Creating Objects. The example program is a usable Workplace Shell object that provides the facility to change the normally unchangable style flags of any Desktop object dropped on it. When you drop an object onto the Workplace Shell Styler, WPSTYLER will open the Settings notebook and display a Style settings page. On this page, you will see the current styles of the object and be able to change them. Using the Workplace Shell Styler, you can change the title in addition to the following behaviors: Copy, Delete, Drag, Drop, Move, Print, Rename, Shadow and Template.

The class name is called WPStyler and is derived from WPAbstract, which is derived from the WPObject root Workplace Shell Class. The implementation of the WPStyler object consists primarily of overrides to instance and class methods inherited from the WPObject and WPAbstract classes.

WPStyler Instance Methods

WPStyler uses ten instance methods:

- wpInitData
- wpDragOver
- wpDrop
- wpOpen
- wpAddSettingsPages
- wpAddObjectWindowPage
- wpAddObjectGeneralPage
- InsertObjectStylePage
- QueryObjectStyle
- SetObjectStyle.

The three last ones are new instance methods introduced by WPStyler.

The following subsections provide explanations on each of them, such as why the instance method is used and whether or not it is overridden.

Initializing Data and Allocating Memory

The wpInitData method is called when an object is created or when it is awakened from the dormant state so that it can initialize all of its instance variables to a known state. Note that this method is called before the object's state is known. Initialize instance data and allocate extra memory that you might need in this method. Any memory allocated in wpInitData should always be d eallocated in wpUnInitData, which is always called when an object is destroyed. WPStyler does not override wpUnInitData.

Dragging Over an Object

The wpDragOver method is invoked on an object as other objects are dragged over it, to determine whether or not it can be dropped on. The wpDragOver method provides information of every object being dragged over the Styler object. The WPStyler will allow all objects to be dropped on and changed.

Dropping an Object

Once an object is dropped onto the Workplace Shell Styler, wpDrop is called. It is, at this point, where WPStyler queries from the drag

information the self pointer of the object being dropped and save it in its instance data. Once the self pointer of the object is acquired, a call to the parent wpOpen is made open the Settings notebook.

Opening a View of an Object

The wpOpen method is called to open a view of this object. The *ulView* parameter tells the object whether to open the Settings view, Contents view, or Help view (and so on), and *param* is an optional parameter. Note that both parameters can be zero, in which case the default view of the object is opened. This instance method is invoked when a user either double clicks on the object or selects the **Settings** item from the pop-up menu. When wpOpen is called as a result of the user selecting the **Open** item or clicking on the object, WPStyler will display a usage message box. OPEN_STYLER is returned as the default view to the **Open** item, because wpQueryDefaultView is subclasses and the Styler's ID is returned as the default view.

Adding the Styler Page to the Settings Notebook

The wpAddSettingsPages method allows each object class to insert its own settings pages into the Settings notebook for an object. WPStyler introduces the InsertObjectStylePage instance method. InsertObjectStylePage is called from the wpAddSettingsPages override to insert the new dialog page into the existing Workplace Shell Settings notebook. InsertObjectStylePage simply fills the PAGEINFO data structure to point to the Styler dialog window and procedure, and it then inserts that page into the notebook by calling wpInsertSettingsPage. InsertObjectStylePage is called only if the notebook is opened for an object that was dropped on the Styler. This method is not called when displaying the Settings notebook for WPStyler.

Removing the Window Page from the Settings Notebook

The wpAddObjectWindowPage method adds the standard Window page to the Settings notebook. WPStyler overrides this method to remove the Window page from the Settings notebook. All WPAbstract objects have a Window page, but as it has no meaning for WPStyler, it will return SETTINGS_PAGE_REMOVED so that this page is not inserted when the notebook is created.

Removing the General Page from the Settings Notebook

The wpAddObjectGeneralPage method adds the standard Icon page to the Settings notebook. This method is overridden so that the Icon page is removed when displaying the styles of an object. This way, when displaying the Settings notebook for the Styler, only its parent needs to be called.

Obtaining the Current Style

QueryObjectStyle is a new instance method introduced by WPStyler. QueryObjectStyle calls wpQueryStyle to query the current style of the object that was dropped onto the Styler, and it then sets the Styler page checkbox and title text to reflect the current state of the object.

Setting the Object's Style

SetObjectStyle is a new instance method introduced by WPStyler. SetObjectStyle queries the current state of the dialog checkboxes and then sets the style of the dropped object accordingly, using wpSetStyle to set the style of the object and wpSaveDeferred to save that state permanently back to the system .INI file. SetObjectStyle also queries and sets the title text of the object.

WPStyler Class Methods

WPStyler uses six class methods:

- wpclsInitData
- wpclsQueryIcon
- wpclsQueryStyle
- wpclsQueryDefaultHelp
- wpQueryDefaultView
- clsQueryModuleHandle.

The last one is a new class method introduced by WPStyler.

The following subsections provide explanations on each of them, such as why the class method is used and whether or not it is overridden.

Initializing Data and Allocating Memory

The wpclsInitData method is called when an object class is instantiated. WPStyler first calls its parent, and it then locates the class file by class name so that it can query the module handle and save it as class data. WPStyler also loads and saves a pointer to its Desktop icon. If any class data is allocated in wpclsInitData, then you should override wpclsUnInitData to free that data when the class is destroyed.

Obtaining the Default Icon

The wpclsQueryIcon method sets the default icon to the Styler icon and resource linked into the application. The pointer handle queried during the creation of the Styler object is returned.

Obtaining the Class Style

The wpclsQueryStyle method is called to return the class attributes for this object class:

Class Style Name	**Description**
CLSSTYLE_NEVERCOPY	The object cannot be copied.
CLSSTYLE_NEVERDELETE	The object cannot be deleted.
CLSSTYLE_NEVERMOVE	The object cannot be moved.

CLSSTYLE_NEVERTEMPLATE	This class does not have a template. Also, the Styler object will not have a **Create another** item in its context menu.

Classes should override this method to get special behavior.

Setting the Default Help

Override wpclsQueryDefaultHelp to set the help library name and default help ID.

Setting the Default View

WPStyler overrides wpQueryDefaultView to set the default view for the Styler object. By returning OPEN_STYLER as the default view, a request to **Open** the object will cause wpOpen to be called with OPEN_STYLER.

Obtaining the Module Handle

The clsQueryModuleHandle method returns the module handle for the WPStyler class. To avoid using globals, the WPStyler class' module handle is saved as part of its class data. This means that a new class method needs to be implemented, so that the new instance methods can gain access to it.

WPStyler Dialog Window Procedure

DialogProc is registered with the dialog window when the Styler page is inserted into the Settings notebook. The dialog procedure gets control when the Styler page is given focus.

The dialog procedure catches all WM_FOCUSCHANGE messages to provide an information message for each control on the Styler settings page. As the user tabs between each control on the Styler page, an information message describing the control is displayed just above the push buttons.

An Apply push button is provided so that a user can apply the changes instantly without closing the Styler page.

WPStyler Functions

WPStyler uses two MLE functions:

- MLEImportText: This function imports text into an MLE window.
- MLEExportText: This function exports text from an MLE window.

Installing Workplace Shell Objects

Workplace Shell objects can be installed on the Desktop in two ways:

- By running an installation program or batch file
- By using the Workplace Shell Class Object List Utility.

Running an Installation Program

You can provide installation programs for your objects. An installation program is responsible for:

- Copying the DLL that contains the object's class definition from a diskette to the \OS2\DLL directory or to a directory in the LIBPATH.
- Registering the class and its DLL name with the Workplace Shell by calling WinRegisterObjectClass.
- Creating an object instance of the class and placing it on the Desktop or in a particular folder by calling WinCreateObject.

An example of an installation program for a Workplace Shell object is shown in the following sample code:

```
/* Command-line program to install Workplace Shell objects  */

#define INCL_WINWORKPLACE
#include <os2.h>
#include <stdio.h>
#include <string.h>

#if defined(DEBUG)
   #define LOCATION_DESKTOP ((PSZ)"<WP_DESKTOP>")
#endif

/*
 *  Main Function
 *
 *    argv[1] = Class Name
 *    argv[2] = Module (DLL) Name
 *    argv[3] = Object Title
 *    argv[4] = Location
 *    argv[5] = Setup String
 */

INT main (argc, argv)
   INT argc;
   CHAR *argv[];

{
  HAB vhab;
  HMQ vhmq;
  BOOL fSuccess;

  if (argc == 1)
  {
    #if defined(DEBUG)
    {
      printf("Usage:\n\n");
      printf("   WPCREATE ClassName ModuleName Title [[Location]
             [SetupString]]\n");
    }
    #endif
    return (0);
  } /* End if (argc == 1) */

  if (argc < 4) return (1);   /* First three parms are mandatory */

  /* Register the class  */
  #if defined(DEBUG)
    printf("WinRegisterObjectClass(%s, %s)...\n", argv[1], argv[2]);
  #endif

  fSuccess = WinRegisterObjectClass(
               argv[1],    /* Class name (case sensitive) */
               argv[2]);   /* Module name                 */
```

```
  if (!fSuccess) return (1);   /* Return non-zero for error */

  #if defined(DEBUG)
    printf("Success: rc = %u\n", fSuccess);
  #endif

  /*  Create an instance of the object  */
  #if defined(DEBUG)
    printf("WinCreateObject(%s, %s,...)...\n", argv[1], argv[3]);
  #endif
  fSuccess = WinCreateObject(
                argv[1],                              /* Class name   */
                argv[3],                              /* Object title */
                argc > 5 ? argv[5] : " ",             /* Setup string */
                argc > 4 ? argv[4] : LOCATION_DESKTOP,/* Location     */
                CO_FAILIFEXISTS);                     /* Flags        */

  if (!fSuccess) return (1);   /* Return non-zero for error */

  #if defined(DEBUG)
    printf("Success: rc = %u\n", fSuccess);
  #endif

  return(0);

}   /* End main() */
```

Instantiating an object is an optional responsibility of an installation program. When a class is registered by calling WinRegisterObjectClass, an object template is placed in the Templates folder on the Desktop, if the class supports templating. Users can create instances of these objects by tearing off a copy of the template. This can be useful for larger applications that define data-file objects that are associated with program objects.

Running an Installation Batch File

An installation batch file written in the REXX language performs the same operations but uses the following REXX-language utility functions:

REXX Functions	**Workplace Shell Methods**
SysCreateObject	WinCreateObject
SysRegisterObjectClass	WinRegisterObjectClass
SysDeRegisterObjectClass	WinDeregisterObjectClass

An example of an installation batch file written using the REXX-language utility methods is shown in the following sample code:

```
/* Register a Workplace Shell class and create an instance */

/* Load the REXX utility functions */
call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
call SysLoadFuncs

/* Register the class with the Workplace Shell */
if SysRegisterObjectClass("NewObjectClass", "NEWDLL") then
  say 'Class Registration successfully completed for the NewObject'

/* Create an instance of the object */
if SysCreateObject("NewObjectClass",
                   "A New Object",
                   "<WP_Desktop>",
                   "OBJECTID=<A New Object>") then
     say 'A New Object successfully created and placed on Desktop'
```

Using the Workplace Shell Class List Object Utility

The Toolkit provides a Workplace Shell Class List Object that is automatically installed during the installation of the Toolkit. This object is a tool that provides a windowed user interface for general Workplace Shell class registration and object creation activities. It performs all the methods that a typical Workplace Shell object installation program must provide, with the exception of copying files from an installation diskette to a hard disk. It is a fast and easy way to build and test objects in an application development environment.

The Workplace Shell Class List Object Utility displays a hierarchical list of all classes registered with the Workplace Shell. You can add a class to this list or perform a number of actions on a specific class in the list. You can also create an instance of the class, replace and unreplace the class, and delete the class.

The Workplace Shell Class List Object Utility uses WinEnumObjectClasses to get the list of all classes registered with the Workplace Shell. WinEnumObjectClasses returns only the name of the class and the DLL module that contains the class definition. It does not return information on the class ancestry that can be used to construct a hierarchical list of classes. Because the Workplace Shell Class List Object Utility is a Workplace Shell object, it can call the somParent function to determine parentage of each class and use this information to construct the list.

The Workplace Shell Class List Object Utility uses the following methods:

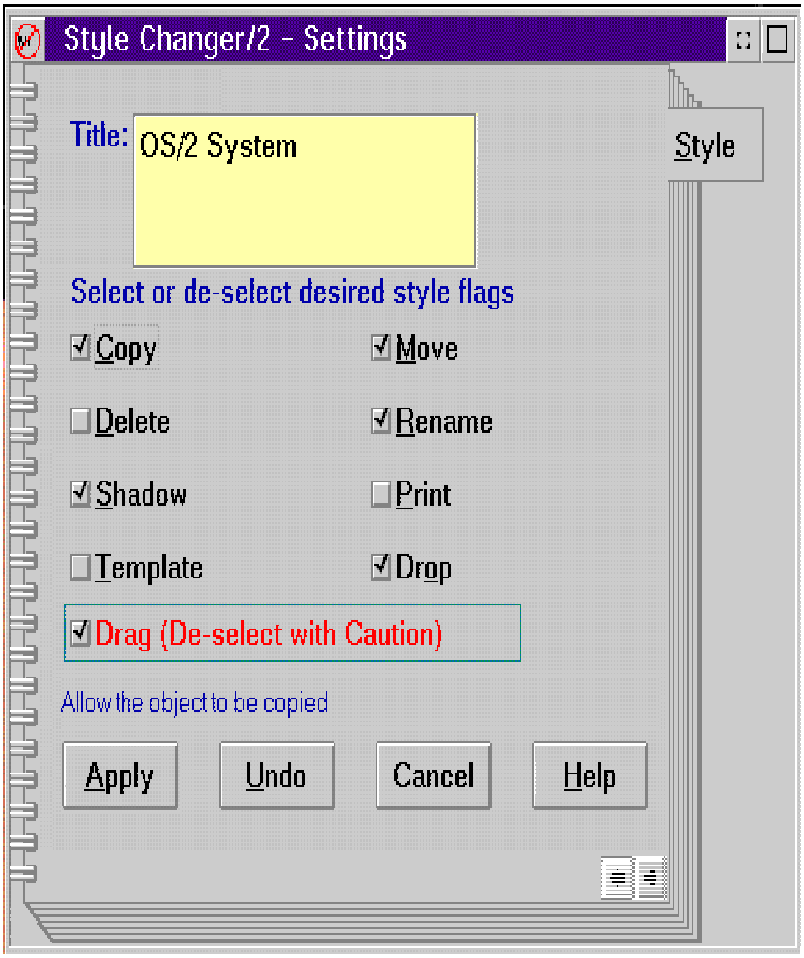
Method Name	**Description**
WinCreateObject	To create objects.
WinRegisterObjectClass	To register object classes.
WinDeregisterObjectClass	To deregister object classes.
WinReplaceObjectClass	To replace object classes.

Note: This tool can be used to delete any Workplace Shell class other than the predefined Workplace Shell classes provided with the OS/2 operating system Version 2.0. You should understand Workplace Shell classes and how they are defined so that they can be recovered. Because the general user may not have this level of understanding, you should not distribute this tool with their objects to the users. The recommended way of delivering objects to them is through installation programs or batch files.

Sample Code for Creating Objects

In order to illustrate what is required to create a complete Workplace Shell object that you can use constructively, a complete Workplace Shell SOM program is included here.

The example program is a usable Workplace Shell object that provides the facility to change the normally unchangable style flags of any Desktop object dropped on it. When you drop an object onto the Workplace Shell Styler, WPSTYLER will open the Settings notebook and display a Style settings page, as shown in the following figure:



On this page, you will see the current styles of the object and be able to change them. Using the Workplace Shell Styler, you can change the title in addition to the following behaviors:

- Copy
- Delete
- Drag
- Drop
- Move
- Print
- Rename
- Shadow
- Template.

The Workplace Shell Styler sample demonstrates the basic steps you need to do to create a Workplace Shell object:

- Create a simple Workplace Shell SOM program
- Create a custom Settings notebook page
- Add a custom page to the Settings notebook
- Remove unwanted pages from the Settings notebook
- Open the Settings notebook
- Implement a custom help for the help push button
- Implement the Pickup and Drop operation in your object
- Manipulate object styles and change their title text.

The class name is called WPStyler and is derived from WPAbstract, which is derived from the WPObject root Workplace Shell Class. This means that WPStyler inherits the methods from WPAbstract, which, in turn, inherits all of the instance and class methods from WPObject. The implementation of the WPStyler object consists primarily of overrides to instance and class methods inherited from the WPObject and WPAbstract classes.

Creating Objects Application Sample Code

Several parts of this program are explained in Creating Objects.

The WPStyler application includes the following files:

File Name	Description
WPSTYLER.IDL	Class interface definition
WPSTYLER.C	Source code
WPSTYLER.RCH	Resource header
WPSTYLER.RC	Resource code
WPSTYLER.IPF	Styler help source
WPSTYLER.MAK	Styler makefile for building the application.

Class Definition File for Styler

The following sample illustrates the class interface definition file (IDL):

```
#ifndef  wpstyler_idl
#define  wpstyler_idl
#include <wpabs.idl>
#include <somcls.idl>

interface M_Styler;

interface Styler : WPAbstract
{
   ULONG InsertObjectStylePage(in HWND hwndDlg);
   //  Override: InsertObjectStylePage
   //
   //  Description:
   //    Insert the Styler page into the Settings notebook.

   ULONG QueryObjectStyle(in HWND hwndDlg);
   //  Override: QueryObjectStyle
   //
   //  Description:
   //    Query the object's title and current style.
   //    Set the Styler Settings page checkboxes and MLE.

   VOID  SetObjectStyle(in HWND hwndDlg);
   //  Override: SetObjectStyle
   //
   //  Description:
   //    Query the current checkbox state and set
   //    the object's style accordingly.

   #ifdef __SOMIDL__
   implementation
   {
      releaseorder: InsertObjectStylePage,QueryObjectStyle,SetObjectStyle;

      /*
       * Class modifiers
       */
      functionprefix = Sty_;
      majorversion   = 1;
      minorversion   = 2;
      filestem       = wpstyler;
      metaclass      = M_Styler;
      callstyle      = oidl;
      dllname        = "wpstyler.dll";

      /*
       * Internal instance variables
       */
      BOOL     fGeneralPage;     // Display the General page if true
      ULONG    ulStyle;          // Save the style flags
      WPObject self;             // Save the self pointer of dropped object
```

```
/*
 * Passthru to the .IH file the following:
 */
passthru C_ih =
"#define INCL_PM"
"#define INCL_DOS"
"#define INCL_DEV"
"#define INCL_WPCLASS"
"#define INCL_WINWORKPLACE"

"#include <os2.h>"
"#include <stdlib.h>"
"#include <string.h>"
"#include \"wpstyler.rch\""
"#include <somobj.h>"
""
"#define OPEN_STYLER            (OPEN_USER+1)"
""

"#define MPNULL                 (MPFROMP(NULL))"
"#define MPZERO                 (MPFROMSHORT(0))"
"#define MRTRUE                 (MRFROMSHORT((SHORT) TRUE))"
"#define MRFALSE                (MRFROMSHORT((SHORT) FALSE))"
""
"#define BM_UNCHECKED           0   // Parms in BM_SETCHECK message"
"#define BM_CHECKED             1   // to set or clear the radio buttons"
""

"#define PMWinSetDlgItemChecked(hwnd, id, bCheck)\"
"               (WinSendDlgItemMsg((hwnd), (id), BM_SETCHECK,\"
"               ((bCheck) ? MPFROMSHORT(1) : MPZERO), MPNULL))"
""
"#define PMWinIsDlgItemChecked(hwnd, id)\"
"               (SHORT1FROMMR(WinSendDlgItemMsg((hwnd), (id),\"
"               BM_QUERYCHECK, MPNULL, MPNULL)) != 0)"
""
"#define WinSetDialogFont(hwnd, usId, szFont)\"
"               (WinSetPresParam(WinWindowFromID(hwnd, usId),\"
"               PP_FONTNAMESIZE, (ULONG) strlen(szFont)+1, szFont))"
"";

/*
 * Method modifiers
 */
wpInitData:                     override;
//  Override: wpInitData
//
//  Description:
//    Initialize the state variables.  Allocate any extra
//    memory that WPStyler might need.

wpDragOver:                     override;
//   Method: wpDragOver
//
//   Description:
//     Allow any object to be dropped on the Styler.

wpDrop:                         override;
//   Method: wpDrop
//
//   Description:
//     Query the self pointer of dropped object.
//     Call wpOpen to open the Settings notebook
//     and display the styles.

wpOpen:                         override;
//   Method: wpOpen
//
//   Description:
//     Opens the Settings notebook to display the General page
//     for the Styler object.

wpAddSettingsPages:             override;
//   Method: wpAddSettingsPages
//
//   Description:
//     Add the Styler Settings page to let the user alter
//     the styles of any object that is dropped onto us.

wpAddObjectWindowPage:          override;
//   Method: wpAddObjectWindowPage
//
//   Description:
```

```
      //     Remove the Window page.

      wpAddObjectGeneralPage:           override;
      //   Method: wpAddObjectGeneralPage
      //
      //   Description:
      //     Remove the General page when the Styler
      //     page is displayed.

   };
   #endif /* __SOMIDL__ */
};

interface M_Styler : M_WPAbstract
{
   HMODULE clsQueryModuleHandle();
   //   Method: clsQueryModuleHandle
   //
   //   Description:
   //     Return the module handle.


   #ifdef __SOMIDL__
   implementation
   {
      releaseorder: clsQueryModuleHandle;

      /*
       * Class modifiers
       */
      functionprefix = StyM_;
      majorversion   = 1;
      minorversion   = 2;
      filestem       = wpstyler;
      callstyle      = oidl;
      dllname        = "wpstyler.dll";

      /*
       * Internal instance variables
       */
      HMODULE  hmod;    /* Resource module handle */
      HPOINTER hicon;   /* Icon handle            */

      /*
       * Method modifiers
       */

      wpclsInitData:                   override;
      //   Method: wpclsInitData
      //
      //   Description:
      //     Initalize the class data, query and save the module handle.
      //     Load the icon handle.

      wpclsQueryIcon:                  override;
      //   Method: wpclsQueryIcon
      //
      //   Description:
      //     Set the Stylers icon handle.

      wpclsQueryStyle:                 override;
      //   Method: wpclsQueryStyle
      //
      //   Description:
      //     wpclsQueryStyle is called to allow the class object to
      //     specify the default object class style for its instances.
      //
      //   Remarks:
      //     This method can be called at any time in order to determine the
      //     default style for instances of this class.
      //     This method should be overridden in order to modify the default
      //     object style for instances of this class.

      wpclsQueryDefaultHelp:           override;
      //   Method: wpclsQueryDefaultHelp
      //
      //   Description:
      //     Set the help library name and default help ID.

      wpclsQueryDefaultView:           override;
      //   Method: wpclsQueryDefaultView
      //
      //   Description:
```

```
      //     Return the default view for a new instance of this object.
      //     Tell the system what the Styler's default view is.
   };
   #endif /* __SOMIDL__ */
};
#endif  /* wpstyler_idl */
```

Source Code for Styler

The following sample illustrates the source code (C):

```
/*
 * File.....: WPSTYLER.C
 *
 * Purpose..: Workplace Shell Object Styler.
 *
 * Instance Methods...:
 *            Sty_InsertObjectStylePage()
 *            Sty_QueryObjectStyle()
 *            Sty_SetObjectStyle()
 *            Sty_wpInitData()
 *            Sty_wpDragOver()
 *            Sty_wpDrop()
 *            Sty_wpOpen()
 *            Sty_wpAddSettingsPages()
 *            Sty_wpAddObjectWindowPage()
 *            Sty_wpAddObjectGeneralPage()
 *
 * Class Methods...:
 *            StyM_clsQueryModuleHandle()
 *            StyM_wpclsInitData()
 *            StyM_wpclsQueryIcon()
 *            StyM_wpclsQueryStyle()
 *            StyM_wpclsQueryDefaultHelp()
 *            StyM_wpclsQueryDefaultView()
 *
 * Dialog Procedures and Functions...:
 *            DialogProc()
 *            MLEImportText()
 *            MLEExportText()
 */

#pragma comment(compiler)
#pragma info(nogen)

#define Styler_Class_Source
#define M_Styler_Class_Source
#include "wpstyler.ih"

/*
 * Prototype local functions used
 */
MRESULT EXPENTRY DialogProc(HWND, ULONG, MPARAM, MPARAM);
ULONG   EXPENTRY MLEImportText(HWND, CHAR *, ULONG);
APIRET  EXPENTRY MLEExportText(HWND, PSZ *);

CHAR szHelpLibrary[] = "wpstyler.hlp";

/*
 *  Override: InsertObjectStylePage
 *
 *  Description:
 *    Insert the Style page into the Settings notebook
 */

SOM_Scope ULONG SOMLINK Sty_InsertObjectStylePage(Styler *somSelf,
                                                  HWND hwndDlg)
{
   PAGEINFO  pi;

   /* StylerData *somThis = StylerGetData(somSelf); */
   StylerMethodDebug("Styler", "Sty_InsertObjectStylePage");
```

```
   memset((PCH) &pi, 0, sizeof(PAGEINFO));
   pi.cb                  = sizeof(PAGEINFO);
   pi.hwndPage            = NULLHANDLE;
   pi.usPageStyleFlags    = BKA_MAJOR;
   pi.usPageInsertFlags   = BKA_FIRST;
   pi.pfnwp               = DialogProc;
   pi.resid               = _clsQueryModuleHandle(_Styler);
   pi.dlgid               = DLG_STYLE;
   pi.pszName             = "~Style";
   pi.pCreateParams       = somSelf;
   pi.pszHelpLibraryName  = szHelpLibrary;

   return _wpInsertSettingsPage(somSelf, hwndDlg, &pi);
}

/*
 *  Override: QueryObjectStyle
 *
 *  Description:
 *    Query the object's title and current style.
 *    Set the Styler Settings page checkboxes and MLE.
 */

SOM_Scope ULONG SOMLINK Sty_QueryObjectStyle(Styler *somSelf,
                                             HWND hwndDlg)
{
   StylerData *somThis = StylerGetData(somSelf);
   StylerMethodDebug("Styler", "Sty_QueryObjectStyle");

   /*
    * Query and set the object's name in the notebook page
    */
   MLEImportText(WinWindowFromID(hwndDlg, DLG_OBJECTNAME),
                 _wpQueryTitle(_self), 0);

   /*
    * Query the object's current state
    */
   _ulStyle = _wpQueryStyle(_self);

   PMWinSetDlgItemChecked(hwndDlg, DLG_COPY,
      _ulStyle & OBJSTYLE_NOCOPY ? BM_UNCHECKED : BM_CHECKED);

   PMWinSetDlgItemChecked(hwndDlg, DLG_SHADOW,
      _ulStyle & OBJSTYLE_NOLINK ? BM_UNCHECKED : BM_CHECKED);

   PMWinSetDlgItemChecked(hwndDlg, DLG_DELETE,
      _ulStyle & OBJSTYLE_NODELETE ? BM_UNCHECKED : BM_CHECKED);

   PMWinSetDlgItemChecked(hwndDlg, DLG_DRAG,
      _ulStyle & OBJSTYLE_NODRAG ? BM_UNCHECKED : BM_CHECKED);

   PMWinSetDlgItemChecked(hwndDlg, DLG_DROP,
      _ulStyle & OBJSTYLE_NODROP ? BM_UNCHECKED : BM_CHECKED);

   PMWinSetDlgItemChecked(hwndDlg, DLG_SETTINGS,
      _ulStyle & OBJSTYLE_NOSETTINGS ? BM_UNCHECKED : BM_CHECKED);

   PMWinSetDlgItemChecked(hwndDlg, DLG_MOVE,
      _ulStyle & OBJSTYLE_NOMOVE ? BM_UNCHECKED : BM_CHECKED);

   PMWinSetDlgItemChecked(hwndDlg, DLG_PRINT,
      _ulStyle & OBJSTYLE_NOPRINT ? BM_UNCHECKED : BM_CHECKED);

   PMWinSetDlgItemChecked(hwndDlg, DLG_RENAME,
      _ulStyle & OBJSTYLE_NORENAME ? BM_UNCHECKED : BM_CHECKED);

   PMWinSetDlgItemChecked(hwndDlg, DLG_TEMPLATE,
      _ulStyle & OBJSTYLE_TEMPLATE ? BM_CHECKED : BM_UNCHECKED);

   return _ulStyle;
}

/*
 *  Override: SetObjectStyle
 *
 *  Description:
 *    Query the current checkbox state and set the style of the
 *    object accordingly.
 */

SOM_Scope void SOMLINK Sty_SetObjectStyle(Styler *somSelf,
                                          HWND hwndDlg)
```

```
{
   PSZ pszName;
   StylerData *somThis = StylerGetData(somSelf);
   StylerMethodDebug("Styler", "Sty_SetObjectStyle");

   /*
    * NOCOPY: This object cannot be copied
    */
   if (PMWinIsDlgItemChecked(hwndDlg, DLG_COPY))
     _ulStyle &= ~OBJSTYLE_NOCOPY;
   else
     _ulStyle |= OBJSTYLE_NOCOPY;

   /*
    * NOSHADOW: This object cannot have a shadow created
    */
   if (PMWinIsDlgItemChecked(hwndDlg, DLG_SHADOW))
     _ulStyle &= ~OBJSTYLE_NOLINK;
   else
     _ulStyle |= OBJSTYLE_NOLINK;

   /*
    * NODELETE: This object cannot be deleted
    */
   if (PMWinIsDlgItemChecked(hwndDlg, DLG_DELETE))
     _ulStyle &= ~OBJSTYLE_NODELETE;
   else
     _ulStyle |= OBJSTYLE_NODELETE;

   /*
    * NODRAG: This object cannot be dragged
    */
   if (PMWinIsDlgItemChecked(hwndDlg, DLG_DRAG))
     _ulStyle &= ~OBJSTYLE_NODRAG;
   else
     _ulStyle |= OBJSTYLE_NODRAG;

   /*
    * NODROP: No other object can be dropped on this object.
    *         However, object can be dragged and dropped on other objects.
    */
   if (PMWinIsDlgItemChecked(hwndDlg, DLG_DROP))
     _ulStyle &= ~OBJSTYLE_NODROP;
   else
     _ulStyle |= OBJSTYLE_NODROP;

   /*
    * NODRAG: This object cannot be moved
    */
   if (PMWinIsDlgItemChecked(hwndDlg, DLG_MOVE))
     _ulStyle &= ~OBJSTYLE_NOMOVE;
   else
     _ulStyle |= OBJSTYLE_NOMOVE;

   /*
    * NOPRINT: This object cannot be printed
    */
   if (PMWinIsDlgItemChecked(hwndDlg, DLG_PRINT))
     _ulStyle &= ~OBJSTYLE_NOPRINT;
   else
     _ulStyle |= OBJSTYLE_NOPRINT;

   /*
    * NORENAME: This object cannot be renamed
    */
   if (PMWinIsDlgItemChecked(hwndDlg, DLG_RENAME))
     _ulStyle &= ~OBJSTYLE_NORENAME;
   else
     _ulStyle |= OBJSTYLE_NORENAME;

   /*
    * NOTEMPLATE: This object is a template
    */
   if (PMWinIsDlgItemChecked(hwndDlg, DLG_TEMPLATE))
     _ulStyle |= OBJSTYLE_TEMPLATE;
   else
     _ulStyle &= ~OBJSTYLE_TEMPLATE;

   /*
    * Extract and set the title text
    */
   MLEExportText(WinWindowFromID(hwndDlg, DLG_OBJECTNAME),
                 &pszName);
```

```
   _wpSetTitle(_self, pszName);
   free(pszName);

   /*
    * Set the object's new style and save the state to the .INI file
    */
   _wpSetStyle(_self, _ulStyle);
   _wpSaveDeferred(_self);
}

/*
 *  Override: wpInitData
 *
 *  Description:
 *    Initialize the state variables.  Allocate any extra
 *    memory that WPStyler might need.
 */

SOM_Scope void SOMLINK Sty_wpInitData(Styler *somSelf)
{
   StylerData *somThis = StylerGetData(somSelf);
   StylerMethodDebug("Styler", "Sty_wpInitData");

   _self = (WPObject *) 0;
   parent_wpInitData(somSelf);
}

/*
 *   Method: wpDragOver
 *
 *   Description:
 *     Allow any object to be dropped on the Styler
 */

SOM_Scope MRESULT SOMLINK Sty_wpDragOver(Styler *somSelf,
                                         HWND hwndCnr,
                                         PDRAGINFO pdrgInfo)

{
   BOOL            fSuccess;
   PDRAGITEM       pditem;    // Drag item pointer

   /* StylerData *somThis = StylerGetData(somSelf); */
   StylerMethodDebug("Styler", "Sty_wpDragOver");

   pditem = DrgQueryDragitemPtr(pdrgInfo, 0);
   if (pditem == NULL)
     DosBeep(500, 100);

   fSuccess = DrgVerifyRMF(pditem, "DRM_OBJECT", NULL);
   if (fSuccess)
     return MRFROM2SHORT(DOR_DROP, DO_COPY);
   else
     return MRFROM2SHORT(DOR_NEVERDROP, DO_COPY);
}

/*
 *   Method: wpDrop
 *
 *   Description:
 *     Query the self pointer of the dropped object.
 *     Call wpOpen to open the Settings notebook and display the styles.
 */

SOM_Scope MRESULT SOMLINK Sty_wpDrop(Styler *somSelf,
                                     HWND hwndCnr,
                                     PDRAGINFO pdrgInfo,
                                     PDRAGITEM pdrgItem)
{
   StylerData *somThis = StylerGetData(somSelf);
   StylerMethodDebug("Styler", "Sty_wpDrop");

   /*
    * Query for the dropped object's self pointer.
    */
   _self = OBJECT_FROM_PREC(pdrgItem->ulItemID);

   /*
    * Call the parents wpOpen to display the Settings notebook
    */
   _fGeneralPage = FALSE;
   parent_wpOpen(somSelf, NULLHANDLE, OPEN_SETTINGS, 0L);
```

```
   return parent_wpDrop(somSelf, hwndCnr, pdrgInfo, pdrgItem);
}

/*
 *   Method: wpOpen
 *
 *   Description:
 *     Opens the Settings notebook to display the General
 *     page for the Styler object.
 */

SOM_Scope HWND SOMLINK Sty_wpOpen(Styler *somSelf,
                                  HWND hwndCnr,
                                  ULONG ulView,
                                  ULONG param)
{
   StylerData *somThis = StylerGetData(somSelf);
   StylerMethodDebug("Styler", "Sty_wpOpen");

   switch(ulView)
   {
      case OPEN_SETTINGS:
         _fGeneralPage = TRUE;
         break;

      case OPEN_STYLER:
      {
         /*
          * If the user double clicks on the Styler object, present a
          * message box instead of openning the Settings page.
          */
         WinMessageBox(HWND_DESKTOP,
                       HWND_DESKTOP,
                       "Drop an object on the Workplace Shell Object "
                       "Styler to change the style state of that object.",
                       "Workplace Shell Object Styler",
                       0,
                       MB_OK |
                       MB_APPLMODAL |
                       MB_MOVEABLE);
         break;
      }
      default:
         break;
   }
   return parent_wpOpen(somSelf, hwndCnr, ulView, param);
}

/*
 *   Method: wpAddSettingsPages
 *
 *   Description:
 *     Add the Styler Settings page to let the user alter the styles
 *     of any object that is dropped onto us.
 */

SOM_Scope BOOL SOMLINK Sty_wpAddSettingsPages(Styler *somSelf,
                                              HWND hwndNotebook)
{
   StylerData *somThis = StylerGetData(somSelf);
   StylerMethodDebug("Styler", "Sty_wpAddSettingsPages");

   parent_wpAddSettingsPages(somSelf, hwndNotebook);

   /*
    * Insert the user page into the Settings notebook
    */
   if (!_fGeneralPage)
     _InsertObjectStylePage(somSelf, hwndNotebook);
   return TRUE;
}

/*
 *   Method: wpAddObjectWindowPage
 *
 *   Description:
 *     Remove the Window page
 */

SOM_Scope ULONG SOMLINK Sty_wpAddObjectWindowPage(Styler *somSelf,
                                                  HWND hwndNotebook)
{
   /* StylerData *somThis = StylerGetData(somSelf); */
```

```
   StylerMethodDebug("Styler", "Sty_wpAddObjectWindowPage");

   return SETTINGS_PAGE_REMOVED;
}

/*
 *   Method: wpAddObjectGeneralPage
 *
 *   Description:
 *     Remove the General page when displaying the Styler page
 */

SOM_Scope ULONG SOMLINK Sty_wpAddObjectGeneralPage(Styler *somSelf,
                                                   HWND hwndNotebook)
{
   StylerData *somThis = StylerGetData(somSelf);
   StylerMethodDebug("Styler","Sty_wpAddObjectGeneralPage");

   return _fGeneralPage ?
          parent_wpAddObjectGeneralPage(somSelf, hwndNotebook) :
          SETTINGS_PAGE_REMOVED;
}

/*
 *   Method: wpclsQueryDefaultHelp
 *
 *   Description:
 *     Set the help library name and default help ID
 */

SOM_Scope BOOL SOMLINK StyM_wpclsQueryDefaultHelp(M_Styler *somSelf,
                                                  PULONG pHelpPanelId,
                                                  PSZ pszHelpLibrary)
{
    /* M_StylerData *somThis = M_StylerGetData(somSelf); */
    M_StylerMethodDebug("M_Styler","StyM_wpclsQueryDefaultHelp");

    if (pHelpPanelId)
      *pHelpPanelId = ID_HELP_STYLER;

    if (pszHelpLibrary)
      strcpy(pszHelpLibrary, szHelpLibrary);

    return TRUE;
}

/*
 *   Method: clsQueryModuleHandle
 *
 *   Description:
 *     Return the module handle
 */

SOM_Scope HMODULE SOMLINK StyM_clsQueryModuleHandle(M_Styler *somSelf)
{
   M_StylerData *somThis = M_StylerGetData(somSelf);
   M_StylerMethodDebug("M_Styler", "StyM_clsQueryModuleHandle");

   return (HMODULE) _hmod;
}

/*
 *   Method: wpclsInitData
 *
 *   Description:
 *     Initalize the class data, query and save the module handle.
 *     Load the icon handle.
 */

SOM_Scope void SOMLINK StyM_wpclsInitData(M_Styler *somSelf)
{
   PSZ     psz;
   somId   stylerId;

   M_StylerData *somThis = M_StylerGetData(somSelf);
   M_StylerMethodDebug("M_Styler", "StyM_wpclsInitData");

   parent_wpclsInitData(somSelf);

   stylerId = somIdFromString("Styler");
   psz = _somLocateClassFile(SOMClassMgrObject,
                             stylerId,
                             Styler_MajorVersion,
```

```
                              Styler_MinorVersion);
   SOMFree(stylerId);

   if (psz != NULL)
     DosQueryModuleHandle(psz, &_hmod);

   /*
    * Load the icon and store it in the class data
    */
   _hicon = WinLoadPointer(HWND_DESKTOP, _hmod, ID_OBJECTICON);
}

/*
 *   Method: wpclsQueryIcon
 *
 *   Description:
 *     Set the Styler's icon handle
 */

SOM_Scope HPOINTER SOMLINK StyM_wpclsQueryIcon(M_Styler *somSelf)
{
   M_StylerData *somThis = M_StylerGetData(somSelf);
   M_StylerMethodDebug("M_Styler", "StyM_wpclsQueryIcon");

   return _hicon;
}

/*
 *   Method: wpclsQueryStyle
 *
 *   Description:
 *     wpclsQueryStyle is called to allow the class object to
 *     specify the default object class style for its instances.
 *
 *   Remarks:
 *     This method can be called at any time in order to determine the
 *     default style for instances of this class.
 *     This method should be overridden in order to modify the default
 *     object style for instances of this class.
 */

SOM_Scope ULONG SOMLINK StyM_wpclsQueryStyle(M_Styler *somSelf)
{
   /* M_StylerData *somThis = M_StylerGetData(somSelf); */
   M_StylerMethodDebug("M_Styler","StyM_wpclsQueryStyle");

   return parent_wpclsQueryStyle(somSelf) | CLSSTYLE_NEVERTEMPLATE;
}

/*
 *   Method: wpclsQueryDefaultView
 *
 *   Description:
 *     Return the default view for a new instance of this object.
 *     Tell the system what the Styler's default view is.
 */

SOM_Scope ULONG SOMLINK StyM_wpclsQueryDefaultView(M_Styler *somSelf)
{
   /* M_StylerData *somThis = M_StylerGetData(somSelf); */
   M_StylerMethodDebug("M_Styler", "StyM_wpclsQueryDefaultView");

   return OPEN_STYLER;
}

typedef struct _WINDATA
{
   SOMAny     *somSelf;
   StylerData *somThis;
   BOOL       fClose;
} WINDATA, *PWINDATA;

/*
 *   Dialog window Procedure
 */

MRESULT EXPENTRY DialogProc(HWND hwndDlg,
                            ULONG msg,
                            MPARAM mp1,
                            MPARAM mp2)
{
   MRESULT  mresultWpRtnCd = MRFALSE;
   PWINDATA pwin = (PWINDATA) WinQueryWindowPtr(hwndDlg, QWL_USER);
```

```
switch(msg)
{
   case WM_INITDLG:
   {
      pwin = (PWINDATA) _wpAllocMem((SOMAny *) mp2,
                                    sizeof(WINDATA), NULL);
      WinSetWindowPtr(hwndDlg, QWL_USER, pwin);

      /*
       * Initialize the WINDATA data structure
       */
      pwin->somSelf     = (SOMAny *) mp2;
      pwin->somThis     = StylerGetData(pwin->somSelf);
      pwin->fClose      = FALSE;

      /*
       * Query the style of the object and set the checkboxes
       */
      _QueryObjectStyle(pwin->somSelf, hwndDlg);

      /*
       * Set the status line font size
       */
      WinSetDialogFont(hwndDlg, DLG_STATUS, "8.Helv");

      /*
       * Return TRUE to tell PM that focus has changed
       */
      mresultWpRtnCd = (MRESULT) TRUE;
      break;
   }  /* End of case WM_INITDLG */

   case WM_FOCUSCHANGE:
   {
      HWND   hwndFocus  = HWNDFROMMP(mp1);

      mresultWpRtnCd = WinDefDlgProc(hwndDlg, msg, mp1, mp2);

      /*
       * When mp2 is FALSE, then hwndFocus is the
       * control window that is receiving focus
       */
      if (SHORT1FROMMP(mp2))
        break;

      /* Title text MLE */
      if (hwndFocus == WinWindowFromID(hwndDlg, DLG_OBJECTNAME))
      {
         WinSetDlgItemText(hwndDlg,
                           DLG_STATUS,
                           "Change the title of the object");
         break;
      }
      else
      /* Apply push button */
      if (hwndFocus == WinWindowFromID(hwndDlg, DLG_APPLY))
      {
         WinSetDlgItemText(hwndDlg,
                           DLG_STATUS,
                           "Save all changes to the system .INI file");
         break;
      }
      else
      /* Undo push button */
      if (hwndFocus == WinWindowFromID(hwndDlg, DLG_UNDO))
      {
         WinSetDlgItemText(hwndDlg,
                           DLG_STATUS,
                           "Undo all changes made since last apply");
         break;
      }
      else
      /* Cancel push button */
      if (hwndFocus == WinWindowFromID(hwndDlg, DLG_CANCEL))
      {
         WinSetDlgItemText(hwndDlg,
                           DLG_STATUS,
                           "Cancel processing the current object");
         break;
      }
      else
      /* Help push button */
```

```
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_HELP))
{
   WinSetDlgItemText(hwndDlg,
                     DLG_STATUS,
                     "Display the user help for Styler");
   break;
}
else
/* Copy checkbox */
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_COPY))
{
   WinSetDlgItemText(hwndDlg,
                     DLG_STATUS,
                     "Allow the object to be copied");
   break;
}
else
/* Delete checkbox */
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_DELETE))
{
   WinSetDlgItemText(hwndDlg,
                     DLG_STATUS,
                     "Make object deletable");
   break;
}
else
/* Drag checkbox */
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_DRAG))
{
   WinSetDlgItemText(hwndDlg,
                     DLG_STATUS,
                     "Allow the object to be dragged");
   break;
}
else
/* Drop checkbox */
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_DROP))
{
   WinSetDlgItemText(hwndDlg,
                     DLG_STATUS,
                     "Allow objects to be drop on object");
   break;
}
else
/* Settings checkbox */
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_SETTINGS))
{
   WinSetDlgItemText(hwndDlg,
                     DLG_STATUS,
                     "Settings checkbox");
   break;
}
else
/* Move checkbox */
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_MOVE))
{
   WinSetDlgItemText(hwndDlg,
                     DLG_STATUS,
                     "Allow this object to be moved");
   break;
}
else
/* Print checkbox */
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_PRINT))
{
   WinSetDlgItemText(hwndDlg,
                     DLG_STATUS,
                     "Make object printable");
   break;
}
else
/* Template checkbox */
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_TEMPLATE))
{
   WinSetDlgItemText(hwndDlg,
                     DLG_STATUS,
                     "Make a template from the object");
   break;
}
else
/* Rename checkbox */
if (hwndFocus == WinWindowFromID(hwndDlg, DLG_RENAME))
{
```

```
      WinSetDlgItemText(hwndDlg,
                        DLG_STATUS,
                        "Allow the object to be renamed");
      break;
   }
   else
   /* Shadow checkbox */
   if (hwndFocus == WinWindowFromID(hwndDlg, DLG_SHADOW))
   {
      WinSetDlgItemText(hwndDlg,
                        DLG_STATUS,
                        "Allow the object to be shadowed");
      break;
   }
   else
   {
      WinSetDlgItemText(hwndDlg, DLG_STATUS, "");
   }

   break;
}  /* End of case WM_FOCUSCHANGE */

case WM_COMMAND:
{
   switch(SHORT1FROMMP(mp1))
   {
      case DLG_APPLY:   /* Apply push button */
      {
         _SetObjectStyle(pwin->somSelf, hwndDlg);

         // Save state to the .INI file now!
         _wpSaveImmediate(pwin->somSelf);

         WinSetDlgItemText(hwndDlg,
                           DLG_STATUS,
                           "Changes applied to object");
         break;
      }

      case DLG_UNDO:   /* Undo push button   */
      {
         _QueryObjectStyle(pwin->somSelf, hwndDlg);
         WinSetDlgItemText(hwndDlg,
                           DLG_STATUS,
                           "Changes undone");
         break;
      }

      case DLG_CANCEL:   /* Cancel push button */
      {
         pwin->fClose = TRUE;
         WinPostMsg(WinQueryWindow(hwndDlg, QW_OWNER),
                    WM_CLOSE, 0L, 0L);
         break;
      }

      case DLG_HELP:   /* Help push button */
      {
         _wpDisplayHelp(pwin->somSelf,
                        ID_HELP_STYLER,
                        szHelpLibrary);
         break;
      }

      default:
         break;
   }
   break;
}  /* End of case WM_COMMAND */

case WM_DESTROY:
{
   /*
    * Set the style of the object from the checkbox settings
    */
   if (!pwin->fClose)
     _SetObjectStyle(pwin->somSelf, hwndDlg);

   _wpFreeMem(pwin->somSelf, (PBYTE) pwin);
   mresultWpRtnCd = WinDefDlgProc(hwndDlg, msg, mp1, mp2);
}  /* End of case WM_DESTROY */

default:
```

```
      {
         mresultWpRtnCd = WinDefDlgProc(hwndDlg, msg, mp1, mp2);
         break;
      } // End of default:
   }
   return mresultWpRtnCd;
}

/*
 *   Function: MLEImportText
 *
 *   Description:
 *     Import the text into an MLE window.
 */

ULONG EXPENTRY MLEImportText(HWND hwndMLE,
                             CHAR *pText,
                             ULONG ulSize)
{
   ULONG  ulLen = ulSize;
   ULONG  cch;
   LINE   lBegin = 0;
   IPT    lOffset = 0;

   WinSendMsg(hwndMLE, MLM_DISABLEREFRESH, 0L, 0L);

   /*
    * Get the current number of characters in the MLE box
    * then send a message to delete them
    */
   cch = (ULONG) WinSendMsg(hwndMLE,
                            MLM_QUERYTEXTLENGTH,
                            0L, 0L);
   WinSendMsg(hwndMLE,
              MLM_DELETE,
              MPFROMLONG(lBegin),
              MPFROMLONG(cch));

   if (pText)
   {
      if (ulLen <= 0L)
        ulLen = (ULONG) strlen(pText);
      WinSendMsg(hwndMLE,
                 MLM_SETIMPORTEXPORT,
                 MPFROMP(pText),
                 MPFROMSHORT((SHORT) ulLen));

      cch = (ULONG) WinSendMsg(hwndMLE, MLM_IMPORT,
                               MPFROMP(&lOffset),
                               MPFROMP(&ulLen));
   }
   WinSendMsg(hwndMLE, MLM_ENABLEREFRESH, 0L, 0L);
   return cch;
} /* End of Function: MLEImportText() */

/*
 *   Function: MLEImportText
 *
 *   Description:
 *     Export the text from an MLE window.
 */
APIRET EXPENTRY MLEExportText(HWND hwndMLE, PSZ *pBuf)
{
   APIRET apiRtnCd = 0;
   LONG   cch;
   LONG   cchnl;
   IPT    lOffset = 0;
   LONG   cbChar = -1;
   CHAR   *ptr;

   *pBuf = (CHAR *) NULL;

   /*
    * Get the length of the data field.
    * Query text length using the selected format.
    */
   cch = (ULONG) WinSendMsg(hwndMLE,
                            MLM_QUERYFORMATTEXTLENGTH,
                            MPFROMLONG(lOffset),
                            MPFROMLONG(cbChar));
   if (cch > 0L)
   {
      if ((ptr = (CHAR *) malloc((USHORT) (cch + 1L))) == (CHAR *) NULL)
```

```
         return PMERR_MEMORY_ALLOCATION_ERR;

      /*
       * Get the text from the MLE window.
       */
      *pBuf = ptr;
      WinSendMsg(hwndMLE,
                 MLM_SETIMPORTEXPORT,
                 MPFROMP(ptr),
                 MPFROMSHORT(cch));
      cchnl = (ULONG) WinSendMsg(hwndMLE,
                                 MLM_EXPORT,
                                 MPFROMP(&lOffset),
                                 MPFROMP((ULONG) &cch));
      *(*pBuf + cchnl) = (CHAR) NULL;
   }
   return apiRtnCd;
} /* End of Function: MLEExportText() */

#pragma info(nouse)
```

Resource Header File for Styler

The following sample illustrates the resource header file (RCH):

```
/*
 * Resouce header module: WPSTYLER.RCH
 */

#define ID_OBJECTICON               255
#define ID_HELP_STYLER              256

#define DLG_STYLE                   100
#define DLG_COPY                    101
#define DLG_DELETE                  102
#define DLG_DRAG                    103
#define DLG_DROP                    104
#define DLG_SETTINGS                105
#define DLG_MOVE                    106
#define DLG_PRINT                   107
#define DLG_TEMPLATE                108
#define DLG_RENAME                  109
#define DLG_SHADOW                  110
#define DLG_OBJECTNAME              111
#define DLG_STATUS                  112
#define DLG_APPLY                   113
#define DLG_UNDO                    114
#define DLG_CANCEL                  115
#define DLG_HELP                    116
```

Resource Code for Styler

The following sample illustrates the resource definition file (RC):

```
/*
 * Resouce module: WPSTYLER.RC
 */

#include <os2.h>
#include "wpstyler.rch"

ICON ID_OBJECTICON WPSTYLER.ICO
```

```
DLGTEMPLATE DLG_STYLE LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG  "", DLG_STYLE, 0, 0, 224, 200, NOT FS_DLGBORDER
    BEGIN
        /*
         * Create a read only MLE window on the page to hold the
         * object's title, similar to the General page title
         */
        LTEXT           "Title:",
                        -1, 10, 133,  24,  8
        MLE             "", DLG_OBJECTNAME,
                        30, 111, 110, 30,
                        MLS_WORDWRAP |
                        WS_GROUP |
                        MLS_IGNORETAB

        /*
         * Place all the Styler checkboxes nicely on the page
         */
        LTEXT           "Select or deselect the desired style flags",
                        -1, 10, 101, 169, 10
        AUTOCHECKBOX    "~Copy",      DLG_COPY,     10,  91,  80, 10
        AUTOCHECKBOX    "~Move",      DLG_MOVE,    106,  91,  80, 10
        AUTOCHECKBOX    "~Delete",    DLG_DELETE,   10,  77,  80, 10
        AUTOCHECKBOX    "~Rename",    DLG_RENAME,  106,  77,  80, 10
        AUTOCHECKBOX    "~Shadow",    DLG_SHADOW,   10,  63,  80, 10
        AUTOCHECKBOX    "~Print",     DLG_PRINT,   106,  63,  80, 10
        AUTOCHECKBOX    "~Template", DLG_TEMPLATE, 10,  49,  80, 10
        AUTOCHECKBOX    "Dr~op",      DLG_DROP,    106,  49,  80, 10

        /*
         * Frame the Drag checkbox and set the text color to red
         * to bring attention to it and to warn of its danger
         */
        CONTROL         "", -1, 8, 36, 146, 11,
                        WC_STATIC,
                        SS_HALFTONEFRAME |
                        WS_GROUP |
                        WS_VISIBLE
        AUTOCHECKBOX    "Drag (De-select with Caution)",
                        DLG_DRAG, 10, 37, 136, 9
                        PRESPARAMS PP_FOREGROUNDCOLOR, 0x00FF0000L

        /*
         * Add a status line just above the push buttons
         */
        LTEXT           "", DLG_STATUS,
                        7, 23, 191, 10, DT_VCENTER

        /*
         * Align the push buttons along the bottom
         */
        PUSHBUTTON      "~Apply",  DLG_APPLY,       7, 7, 38, 14
        PUSHBUTTON      "~Undo",   DLG_UNDO,       57, 7, 38, 14
        PUSHBUTTON      "Cancel",  DLG_CANCEL,    107, 7, 38, 14
        PUSHBUTTON      "~Help",   DLG_HELP,      157, 7, 38, 14
    END
END
```

Help Source for Styler

The following sample illustrates the Styler help source file (IPF):

```
.*****************************************************************
.*  File Name:    WPSTYLER.IPF
.*  Description:  Information Tag Language file for help on the
.*                Workplace Shell Object Styler
.*****************************************************************
.*
:userdoc.
:h1 res=256 name=ID_HELP_STYLER.Workplace Shell Object Styler
```

```
:i1 id=aboutStyle.Workplace Shell Object Styler
:p.
The Workplace Shell Object Styler is a sample Workplace Shell SOM
application which demonstrates the basic functionality of a Workplace
Shell SOM application.
The application allows a user to change and save the basic style
attributes of any object that is dropped on it.
.*
:note.
Use this application with care, deselecting all of the items
(especially the Drag) will prevent you from using WPSTYLER again
in order to reinstate the styles.
.*
:p.
:hp2.Title Bar Text:ehp2.
:p.
You can modify the title bar text of the object from the Workplace
Shell Styler in addition to changing the object style.
Changing the title text here has the same effect as changing it
from the General page of the object itself.
:p.
:hp2.Items:ehp2.
.*
:ul.
:li.
Copy
:p.
Deselect this item to prevent the object from being copied.
The :hp2.Copy...:ehp2. item will be removed from the
pop-up menu.
:li.
Move
:p.
Deselect this item to prevent the object from being moved.
The :hp2.Move...:ehp2. item will be removed from the
pop-up menu.
:li.
Delete
:p.
Deselect this item to prevent the object from being deleted.
The :hp2.Delete...:ehp2. item will be removed from the
pop-up menu.
:li.
Rename
:p.
Deselect this item to prevent the object from being renamed.
No visual change can be observed.
:li.
Shadow
:p.
Deselect this item to prevent the object from being shadowed.
The :hp2.Create shadow...:ehp2. item will be removed
from the pop-up menu.
:li.
Print
:p.
Deselect this item to prevent the object from being printed.
The :hp2.Print...:ehp2. item will be removed from the
pop-up menu.
:p.
The print item is normally only visible on objects that can be printed
such as a document file viewed through the Drives folder.
Therefore, this item will have no effect on objects such as folders.
:li.
Template
:p.
Selecting this item will change the object into a template, at which
point you can only tear off objects from it.
:li.
Drop
:p.
Deselect this item to prevent objects from being dropped onto it.
:li.
Drag
:p.
Deselect this item with care.
This will prevent the object from being dragged; therefore you will
not be able to drag the object over the Workplace Shell Styler again
to change the item back.
:p.
Use this item where you wish to have an object remain on the Desktop
in a certain location and not give users the ability to change that location.
:eul.
```

```
.*
:p.
:hp2.Push buttons:ehp2.
.*
:ul.
:li.
Apply
:p.
Press this button to apply the changes immediately to the object while
remaining in the Workplace Shell Styler dialog window.
:li.
Undo
:p.
Press this button to undo any of the checkboxes changed since the
last apply.
:li.
Cancel
:p.
Press this button to cancel your changes and remain at the state you
started at or at the state of the last apply.
:note.
This is different than closing the window through the :hp2.Close:ehp2.
item on the system menu.
The system :hp2.Close:eph2. item saves the current state of the
checkboxes.
:li.
Help
:p.
This push button present this help panel.
:eul.
.*
:euserdoc.
```

Make File for Styler

The following sample illustrates the make file (MAK):

```
#
# Makefile module: WPSTYLER.MAK
#

.SUFFIXES: .c .obj .dll .idl .h .ih .rc .res

SC        = sc
SCFLAGS   = -maddstar -S128000 -C128000 -mnoint -v
SCLIST    = -s "ih;h;c;def"

TOOLKIT   = c:\toolkit
IBMCPP    = c:\ibmcpp

INCLUDE   = $(TOOLKIT)\h;$(IBMCPP)\include;.;$(INCLUDE)
SMINCLUDE = S(TOOLKIT)\idl;$(INCLUDE);$(SMINCLUDE)
LIB       = $(TOOLKIT)\lib;$(IBMCPP)\lib;$(LIB)
CFLAGS    = -Ge- -Ss+ -C+ -Kb -Q+ -Ms
LFLAGS    = /MAP /NOL /NOI /EXEPACK:2 /PACKCODE /PACKDATA /FAR /ALIGN:4

b=wpstyler

all:        $(b).dll $(b).ih $(b).hlp

.c.obj:     icc $(CFLAGS) $*.c

$(b).dll:   $(b).ih $(b).obj $(b).def $(b).res
            link386 $(b) $(LFLAGS), $(b).dll, $(b).map, somtk, $(b)
            rc $(b).res $(b).dll
            mapsym $(b).map

$(b).obj:   $(b).c $(b).h

$(b).res:   $(b).rc $(b).ico
            rc -r $(b).rc
```

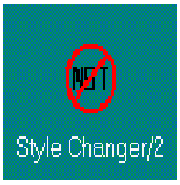
```
$(b).ih:    $(b).idl
            $(SC) $(SCFLAGS) $(SCLIST) $(@B).idl

$(b).hlp:   $(b).ipf
            ipfc $(b).ipf

clean:
            -del $(b).ih  $(b).h  $(b).obj  $(b).dll
            -del $(b).map   $(b).sym  $(b).res  $(b).def
```

Representing the Styler Object

Use the icon editor to create an icon that is suitable for representing the Styler object on the Desktop and save it as WPSTYLER.ICO. The following figure shows a sample icon for the Styler object:



Now you are set to build and register your object with the Workplace Shell.

Debugging Workplace Shell Applications

Debugging a Workplace Shell application is somewhat like debugging a Presentation Manager application. But because Workplace Shell objects are implemented via DLLs instead of EXEs, they can be more of a challenge. This chapter describes the SOM and Workplace Shell facilities and techniques to aid debugging applications.

About SOM Debugging Techniques

Normally when an application is not working correctly, adding temporary printf statements or WinMessageBox calls can provide additional debugging information. SOM provides somPrintf that writes debugging information to the current stdout device. This is usually not convenient for Workplace Shell applications, because they are implemented in DLLs.

To redirect the debugging output to another location, SOM provides a replaceable procedure called SOMOutCharRoutine. In addition to handling calls from somPrintf, the SOMOutCharRoutine is also invoked by the <classname>MethodDebug calls in every method. By replacing this procedure, output can be redirected from stdout to a more suitable location. When debugging is complete, the user-defined procedure can be removed or commented-out.

Debugging information can be written to any number of places:

- To stderr
- To a window
- To the clipboard
- To a log file
- Or any combination of the above.

About Workplace Shell Debugging Techniques

The *Workplace process* is the one under which all the Workplace Shell classes are loaded and initialized. Therefore, objects representing Workplace Shell classes and their subclasses must run on this process. The Workplace process is actually launched from the *Shell process*, which is the process indicated in the SET PROTSHELL= statement in the CONFIG.SYS file. Once the Shell process is running, it starts the Workplace process. It is the Shell process that is responsible for restarting the Workplace process in the event that it is killed as a result of a trap. The PROTSHELL= statement in the CONFIG.SYS file indicates which process is to be launched as the Shell process. The SET RUNWORKPLACE= statement in the CONFIG.SYS file indicates which process is to be the Workplace process. In the default configuration, both the PROTSHELL and RUNWORKPLACE environment variables are set to PMSHELL.EXE. PMSHELL.EXE is designed to distinguish between being started as the Shell process versus being started as the Workplace process.

Using SOM Debugging Techniques

This section describes two SOM techniques to aid debugging Workplace Shell applications:

- One writes the information to a file
- The other writes the information out through a serial port to an attached computer or dumb terminal.

Redirecting SOMOutCharRoutine to a File

The following sample code shows how to direct somPrintf output to a file called C:\ERROR.TXT. Before enabling the replacement of SOMOutCharRoutine, the file is opened and a header is written to it.

```
#define INCL_DOS
#define INCL_WINWORKPLACE

#include <os2.h>
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <string.h>
#include <som.h>

/* File to contain debugging error information */
static FILE *ErrorFile;

/******************************************************************/
/*   This is the replacement for the default SOMOutCharRoutine.   */
/*   It writes the debugging information to a file on the hard drive  */
/*   instead of to stdout.                                        */
/******************************************************************/

#pragma linkage(myCharacterOutputRoutine, system)

int SOMLINK myCharacterOutputRoutine(char chOut)
{
    fputc((int) chOut, ErrorFile);
    fflush(ErrorFile);

    return 1;   /* Indicate success */
}
   .
   .
   .

/******************************************************************/
/*   Enable myCharacterOutputRoutine by opening the output error  */
/*   file and then changing the output character routine.         */
/******************************************************************/

ErrorFile = fopen ("c:\\error.txt", "a");

fprintf (ErrorFile, "\nDebug information from my WPS Object\n");
fprintf (ErrorFile,   "-----------------------------------\n");
setbuf  (ErrorFile, NULL);
```

```
SOM_TraceLevel  = 2;   /* Request maximum debugging information */
SOM_WarnLevel   = 2;
SOM_AssertLevel = 2;

/* All output goes to my routine after the next statement */

SOMOutCharRoutine = myCharacterOutputRoutine;
```

Redirecting SOMOutCharRoutine to a Serial Port

Instead of directing the output to a file, the SOMOutCharRoutine procedure can be used to direct the output to a serial port. By connecting another computer or a dumb terminal to the serial port with a NULL modem cable, debugging information can be received on the remote terminal. A NULL modem cable is a specialized serial cable that has the transmit and receive wires crossed so that transmissions from one serial port are received by another.

First, initialize the serial port using the OS/2 **Mode** command. This command can be added to the STARTUP.CMD file or entered from an OS/2 command prompt. The following example shows how to use the **Mode** command:

```
MODE COM1 9600,n,8,1
```

Replace SOMOutCharRoutine with your procedure which directs somPrintf output to the COM1 serial port as shown in the following sample code:

```
#define INCL_DOS
#define INCL_WINWORKPLACE

#include <os2.h>
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <string.h>
#include <som.h>

/* COM1 port for debugging information */
static FILE *DebugPort;

/*********************************************************************/
/*   This is the replacement for the default SOMOutCharRoutine.      */
/*   It writes the debugging information to the COM1 serial port     */
/*   instead of to stdout.                                           */
/*********************************************************************/

#pragma linkage(myCharacterOutputRoutine, system)

int SOMLINK myCharacterOutputRoutine(char chOut)
{
    fputc((int) chOut, DebugPort);
    fflush(DebugPort);

    return 1;   /* Indicate success */
}
   .
   .
   .

/*********************************************************************/
/*   Enable myCharacterOutputRoutine by opening the serial port      */
/*   and then changing the output character routine.                 */
/*********************************************************************/

DebugPort = fopen("COM1", "w");

fprintf(DebugPort, "\nDebug information from my WPS Object\n");
fprintf(DebugPort,   "----------------------------------\n");
setbuf(DebugPort, NULL);
```

```
SOM_TraceLevel  = 2;   /* Request maximum debugging information */
SOM_WarnLevel   = 2;
SOM_AssertLevel = 2;

/* All output goes to my routine after the next statement */

SOMOutCharRoutine = myCharacterOutputRoutine;
```

Using Workplace Shell Debugging Techniques

This section describes how to use the PROTSHELL and RUNWORKPLACE variables to setup a development environment. It also provides two techniques to aid debugging Workplace Shell applications:

- Lower level debugger, the Kernel Debugger
- Source level debugger, IPMD

Debugging with Useful CONFIG.SYS Statements

The PROTSHELL= statement specifies the name of the program for the Shell process of the Workplace Shell. The default is PMSHELL.EXE. This program loads the Workplace process and handles restarting it, should it fail.

The SET RUNWORKPLACE= statement specifies the name of the program for the Workplace process of the Workplace Shell. The default is also PMSHELL.EXE. PMSHELL recognizes whether it should act as the Shell process or as the Workplace process. Setting this to be CMD.EXE will cause an OS/2 command prompt to be started instead of the Workplace Shell process.

The SET SHELLEXCEPTIONHANDLER=OFF statement disables the Workplace Shell trap handler. The default is ON, which can mask problems with a Workplace Shell application.

The SET RESTARTOBJECTS=NO statement prevents objects from being re-opened or re-started when the Workplace Shell is initialized. This is useful when there is an object that is causing a problem in the Workplace Shell.

Starting and Restarting the Workplace Shell

The Workplace Shell starts automatically when OS/2 is booted. Because all Workplace Shell applications run under the Workplace process, one application with an error can cause the Workplace Shell to trap or to hang. Should the Workplace Shell become inoperative, the system must be rebooted. To avoid frequent reboots, one method that can be used is to start the Workplace Shell from an OS/2 window. Then should a hang occur, the Workplace Shell can be terminated by going to the OS/2 window and entering a Ctrl+Break.

The procedure for manually starting and restarting the Workplace Shell is as follows:

1. Create a backup copy of your CONFIG.SYS file, for example:

   ```
   COPY CONFIG.SYS CONFIG.WPS
   ```

2. Edit CONFIG.SYS. Find the following line:

   ```
   SET RUNWORKPLACE=C:\OS2\PMSHELL.EXE
   ```

 where C: is your boot drive. This line sets the name of the interface to be started for OS/2. PMSHELL.EXE is the program for the Workplace Shell. Replace the line with the following:

   ```
   SET RUNWORKPLACE=C:\OS2\CMD.EXE
   ```

This changes the OS/2 interface from the Workplace Shell to an OS/2 session.

3. By default, the Workplace Shell attempts to recover from any traps caused by itself or other Workplace Shell applications. Should this recovery occur, you might see all the objects on the Desktop disappear temporarily and then re-appear after the Workplace Shell re-initializes. The Workplace Shell exception handler can be disabled by adding the following line to the CONFIG.SYS file:

   ```
   SET SHELLEXCEPTIONHANDLER=OFF
   ```

 With the exception handler disabled, errors will result in the typical OS/2 error pop-up windows. Thus, errors encountered will not be masked by the Workplace Shell's exception handler. However, the ability for the Workplace Shell to recover from errors has been impacted.

4. Save the changes to CONFIG.SYS. These changes will not take effect until a shutdown and reboot are performed.

5. Shutdown the system and then reboot.

6. After the system reboots, instead of bringing up the Workplace Shell (PMSHELL.EXE), the system will bring up an OS/2 prompt (CMD.EXE).

7. Start another OS/2 session by issuing START or START /FS for a full-screen session. This session can be used for running other programs if needed.

8. Return to the original OS/2 session, and then manually start the Workplace Shell by typing:

   ```
   PMSHELL
   ```

It is important to invoke PMSHELL from the first OS/2 session. There are two reasons for this:

- First, the Workplace Shell assumes that it is the first process in the system.
- Secondly, OS/2 assumes that the first application to create a message queue is the Workplace Shell and allocates additional resources for it. If another application creates a message queue before the Workplace Shell, unpredictable problems occur when the Workplace Shell is started.

Now that the Workplace Shell has been started from an OS/2 session, should the Workplace Shell trap or hang, it can be stopped by simply going back to that OS/2 session and terminating the session, for example, by selecting the Ctrl+Break keys. The Workplace Shell can then be restarted from the same window by invoking PMSHELL again.

After the Workplace Shell has terminated, the DLLs containing Workplace Shell objects should be unlocked. This allows them to be deleted or replaced before restarting the Workplace Shell.

Debugging with the Kernel Debugger

The Kernel debugger included with the OS/2 Toolkit is a low-level debugger oriented toward system and device-driver debugging. The kernel debugger can assist in debugging Workplace Shell applications until higher-level debuggers, such as IPMD, can provide adequate debugging functions in complex situations.

When the Kernel debugger is installed, two files are copied to the root directory of the boot drive: OS2KRNL and OS2KRNL.SYM. OS2KRNL is a hidden file, which will be renamed OS2KRNL.RTL when debugging is active. Because the Kernel debugger supports symbolic debugging, numerous symbol files (.SYM files) are copied to the \OS2\DEBUG\DLL directory. These symbol files contain information on symbols defined and used by OS/2.

To make symbols defined in your Workplace Shell application available to the Kernel debugger, use the MAPSYM program. MAPSYM converts the .MAP file generated by the linker to a .SYM file. The .SYM file should reside in the same directory as the .EXE or .DLL executable file it represents and it should have the same name.

When using the Kernel debugger, it is a good idea to use the *functionprefix=* modifier in the IDL file implementation statement. There are three reasons for this:

- All the methods in the class start with the same prefix
- The methods are not static
- It is easier to locate the appropriate methods in the debugger and to set breakpoints.

In the following sample code, the methods generated are called Dogdisplay and Dogbark.

```
/*****************************************************************/
/* By adding the functionprefix= modifier to the class, all      */
/* methods defined will be unique, for example, Dogbark.         */
/* This is useful when debugging, because every method in the    */
/* class will start with this prefix and will not be static.     */
/*****************************************************************/

#ifndef dogdbug_idl
  #define dogdbug_idl

  #include <somobj.idl>

  interface Dog : SOMObject
  {
    attribute string breed;

    // The breed for the dog
    void display();

    // Display characteristics for this dog
    void bark();

    // Have the dog bark
    #ifdef __SOMIDL__
      implementation
    {
      releaseorder:  _get_breed, _set_breed, display, bark;

      //# Class modifiers
      functionprefix = Dog;   // This will help when debugging
      callstyle      = oidl;
      majorversion   = 1;
      minorversion   = 2;

    }; #endif /* __SOMIDL__ */

  }; #endif
```

As soon as the Kernel debugger is active, use the **.p** command to display process information. The Process ID (pid) of the Shell process and the Workplace process can be located quickly using this command. The following figure shows an example of the information you obtain when using the **.p** command:

```
## .p
   Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
   0001  0001 0000 0000 0001 blk 0100 ffe3a000 ffe3c7d4 ffe3c620 1e7c 00 *ager
   0002  0001 0000 0000 0002 blk 0200 7b92a000 ffe3c7d4 7bb28020 1f3c 00 *tsd
   0003  0001 0000 0000 0003 blk 0200 7b92c000 ffe3c7d4 7bb281d4 1f50 00 *ctxh
   0004  0001 0000 0000 0004 blk 081f 7b92e000 ffe3c7d4 7bb28388 1f48 00 *kdb
   0005  0001 0000 0000 0005 blk 0800 7b930000 ffe3c7d4 7bb2853c 1f20 00 *lazyw
   0006  0001 0000 0000 0006 blk 0800 7b932000 ffe3c7d4 7bb286f0 1f3c 00 *asyncr
   0008  0005 0001 0005 0001 blk 0200 7b936000 7bb458a4 7bb28a58      00 cmd
  *0009# 0006 0005 0006 0001                              7bb28c0c 1eb8 01 pmshell
   000d  0006 0005 0006 0002                              7bb292dc      01 pmshell
   000e  0006 0005 0006 0003    In this example,          7bb29490      01 pmshell
   000f  0006 0005 0006 0004    Pid #6 is the first       7bb29644      01 pmshell
   0010  0006 0005 0006 0005    PMSHELL process, or       7bb297f8      01 pmshell
   0007  0006 0005 0006 0006    the "Shell process".      7bb288a4 1ecc 01 pmshell
   0013  0006 0005 0006 0007                              7bb29d14      01 pmshell
   0015  0006 0005 0006 0008                              7bb2a07c      01 pmshell
   0016  0006 0005 0006 0009 blk 0200 7b952000 7bb44020 7bb2a230      01 pmshell
   0017  0006 0005 0006 000a blk 0800 7b954000 7bb44020 7bb2a3e4      01 pmshell
   0018  0006 0005 0006 000b blk 0800 7b956000 7bb44020 7bb2a598      01 pmshell
   0019  0006 0005 0006 000c blk 0800 7b958000 7bb44020 7bb2a74c      01 pmshell
   001a  0006 0005 0006 000d blk 0804 7b95a000 7bb44020 7bb2a900      01 pmshell
   001b  0006 0005 0006 000e blk 0804 7b95c000 7bb44020 7bb2aab4      01 pmshell
   001c  0006 0005 0006 000f blk 0500 7b95e000 7bb44020 7bb2ac68      01 pmshell
   001d  0006 0005 0006 0010 blk 0800 7b960000 7bb44020 7bb2ae1c 1bb0 01 pmshell
   Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
   001e  0006 0005 0006 0011 blk 0800 7b962000 7bb44020 7bb2afd0 1b8c 01 pmshell
   001f  0006 0005 0006 0012 blk 0200 7b964000 7bb44020 7bb2b184      01 pmshell
   000a  0003 0000 0003 0001 blk 0200 7b93a000 7bb4484c 7bb28dc0      00 lanmsgex
   000b  0004 0000 0004 0001 blk 0200 7b93c000 7bb45078 7bb28f74      00 lsdaemon
   000c  0007 0006 0007 0001 blk 0800 7b93e000 7bb460d0 7bb29128      00 harderr
   0011  0007 0006 0007 0002 blk 0800 7b948000 7bb460d0 7bb299ac      00 harderr
   0012  0007 0006 0007 0003 blk 0800 7b94a000 7bb460d0 7bb29b60      00 harderr
   0014  0008 0000 0008 0001 blk 0200 7b94e000 7bb468fc 7bb29ec8      01 stoplan
```

```
   0020  0009 0006 0009 0001 blk 0500 7b966000 7bb47128 7bb2b338 1eb8 10 pmshell
   0021  0009 0006 0009 0002 blk 0200 7b968000 7bb47128 7bb2b4ec      10 pmshell
   0022  0009 0006 0009 0003                           7bb2b6a0 1eb8 10 pmshell
   0023  0009 0006 0009 0004                           7bb2b854 1ecc 10 pmshell
   0024  0009 0006 0009 0005    In this example,       7bb2ba08      10 pmshell
   0025  0009 0006 0009 0006    Pid #9 is the second   7bb2bbbc      10 pmshell
   0026  0009 0006 0009 0007    PMSHELL process, or    7bb2bd70      10 pmshell
   0027  0009 0006 0009 0008    the "Workplace         7bb2bf24      10 pmshell
   0028  0009 0006 0009 0009    process".              7bb2c0d8      10 pmshell
   002a  0009 0006 0009 000b                           7bb2c440 1eac 10 pmshell
   002b  0009 0006 0009 000c                           7bb2c5f4 1eb8 10 pmshell
   0029  000a 0006 000a 0001 blk 0200 7b978000 7bb47954 7bb2c28c      11 cmd
   002c  000b 0006 000b 0001 blk 0200 7b97e000 7bb48180 7bb2c7a8      12 cmd
   002f  001f 0006 001f 0001 blk 0200 7b984000 7bb4a230 7bb2ccc4      15 cmd
   Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
   0030  001d 0006 001d 0001 blk 0200 7b986000 7bb49a04 7bb2ce78      13 cmd
##
```

The Kernel debug command **VSF *** sets the trap vectors. This allows traps in the Workplace Shell application to be located quickly. The Kernel debugger stops on the instruction that is about to cause a trap when these vectors are set. In this way, it is possible to examine the stack and registers to determine the cause of the trap.

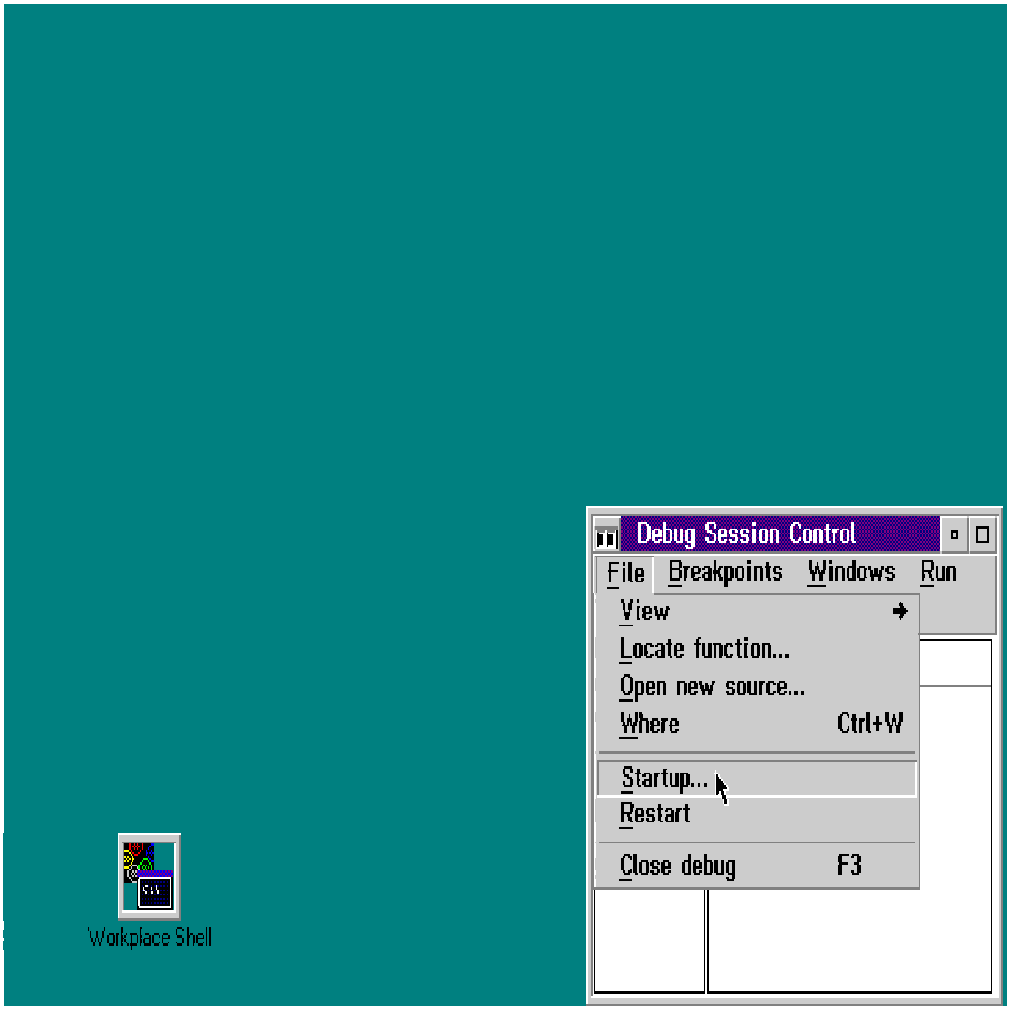
Debugging with IPMD

The Starting and Restarting the Workplace Shell technique can also be used when a source level debugger is needed. Follow the same steps as outlined in that section, but instead of simply invoking PMSHELL, invoke the debugger, as follows:

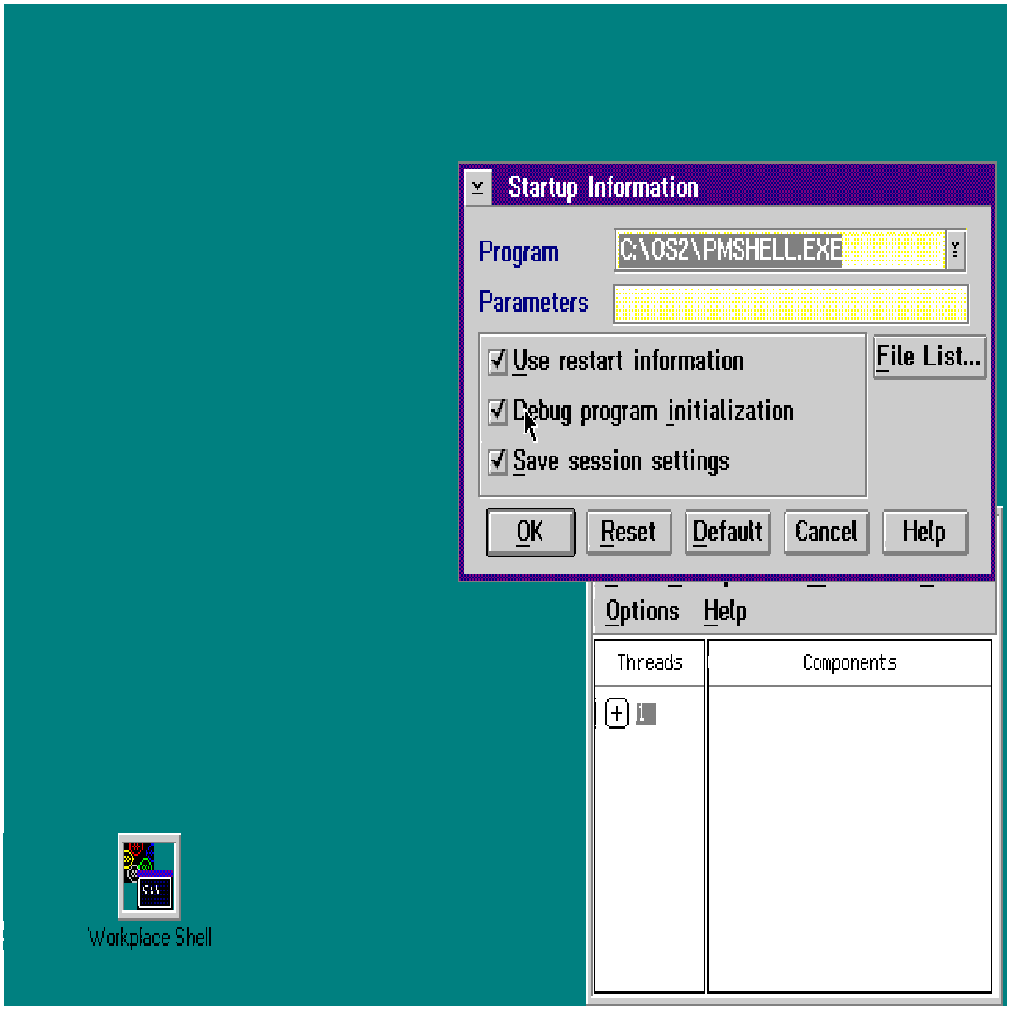
```
IPMD PMSHELL
```

The debugger initializes, but the Workplace Shell does not start running. The steps to be followed are:

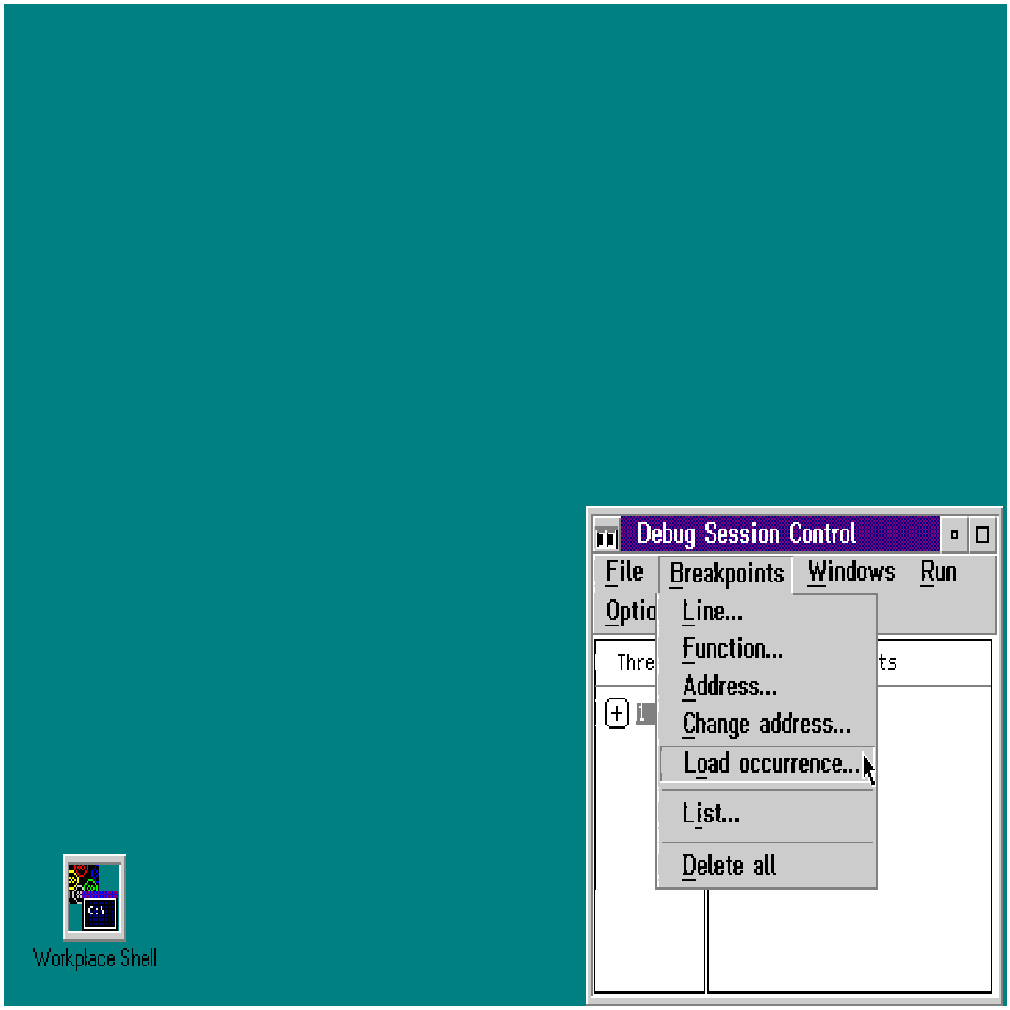
1. Select the **Startup** option from the **File** pull-down menu, as shown in the following figure:



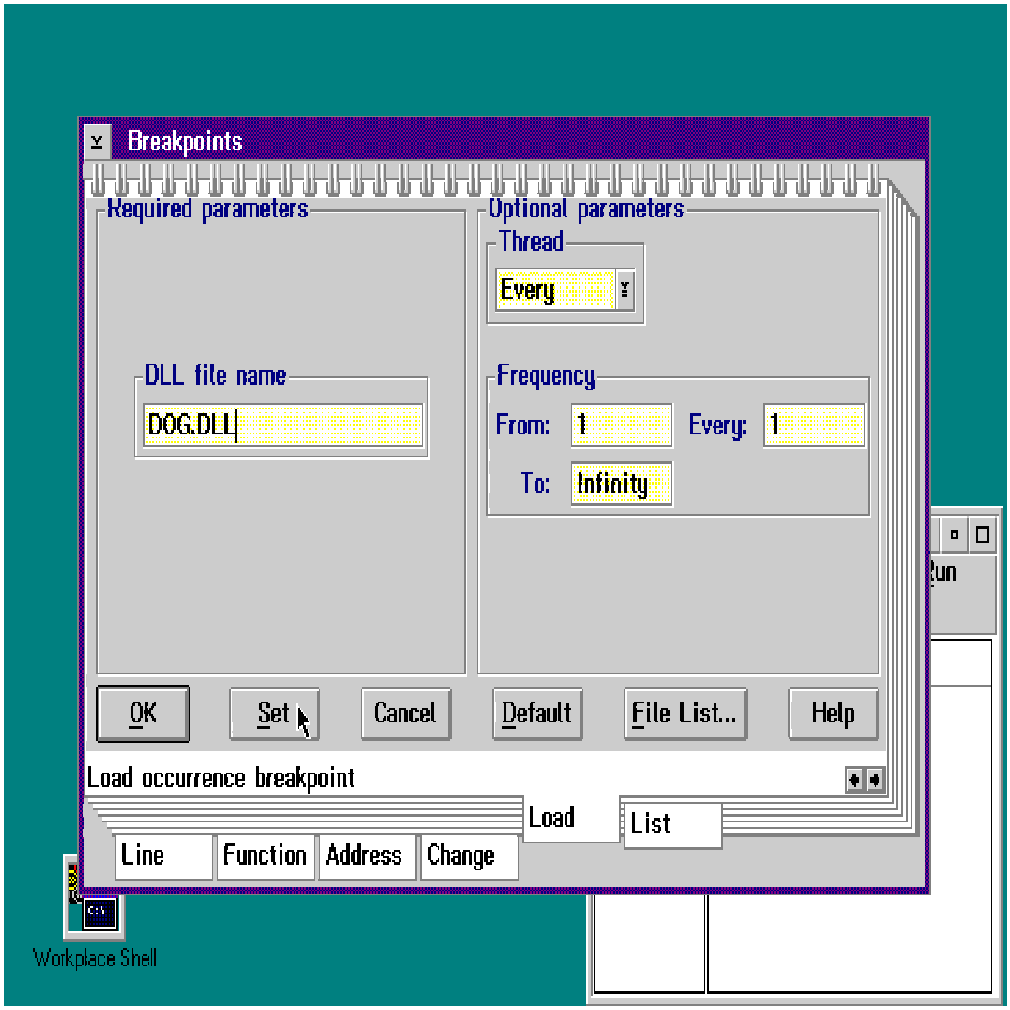
2. Select the Debug program initialization option, as shown in the following figure:



3. Select the **Load Occurence** option from the **Breakpoints** pull-down menu, as shown in the following figure:

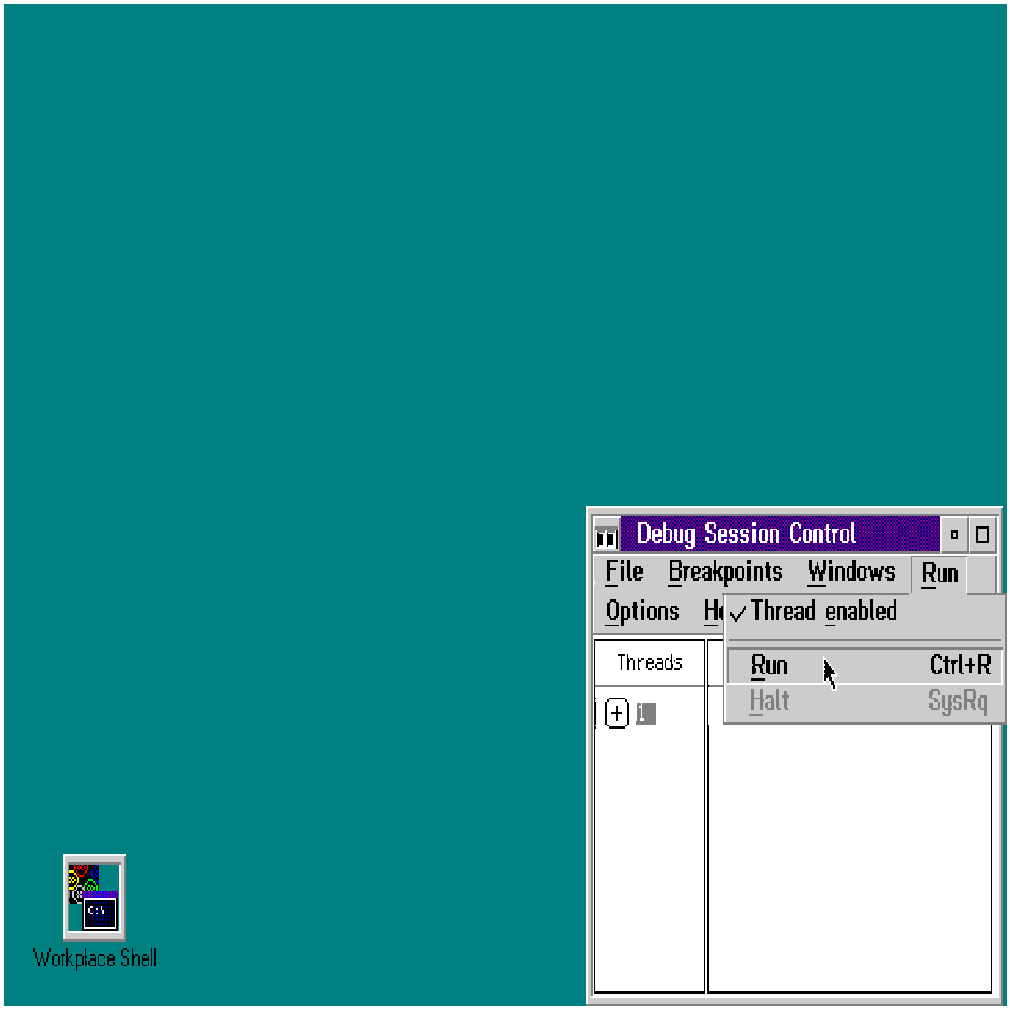


4. Enter a breakpoint by specifying the name (but not the path) of the DLL, for example, DOG.DLL. Then, select the Set push button, as shown in the following figure:



5. Set any other breakpoints you wish at this time. Select the OK push button when finished.

6. Select the **Run** option from the **Breakpoints** pull-down menu, as shown in the following figure:



The Workplace Shell will then initialize (very slowly) and the debugger will stop when one of the specified breakpoints is reached.

Debugging a Workplace Shell application using IPMD will be much the same as debugging a PM application. Remember that if you manually enter the name of a class or instance variable into IPMD, you must prefix the name with *somThis* instead of using the underscore (_) macro. In the sample code fragment illustrated in the following figure, to manipulate the *_hIcon* variable in IPMD, use *somThis hIcon* instead of *_hIcon* (selecting *_hIcon* with the mouse works as expected).

```
/************************************************/
/*  Method: wpclsInitData                       */
/*                                              */
/*  This sample shows how to initialize class data  */
/ *  associated with MyDataBase.                 */
/************************************************/

SOM_Scope void SOMLINK MyDataBaseM_wpclsInitData(M_MyDataBase *somSelf)
{
  /* M_MyDataBaseData *somThis = M_MyDataBaseGetData(somSelf); */
  M_MyDataBaseMethodDebug("M_MyDataBase","MyDataBaseM_wpclsInitData");

  hModule = _clsQueryModuleHandle(somSelf);

  _hIcon = WinLoadPointer(HWND_DESKTOP, hModule, ID_WINDOW);

  parent_wpclsInitData(somSelf);
}
```

Object Aid: Help Methods

This chapter describes how to write Workplace Shell applications that provide help information.

About Help Methods

While running an application or viewing a document, the user may require help on the application or document. For example, an application may be a descendant of WPFolder class, to which you added a new context menu. You then need to provide a help dialog for this new context menu. It is important to include the help information for all applications as it is used often. This chapter describes different methods that can be overridden in order to provide help from an application window when a user requests it either by:

- Pressing the F1 key
- Selecting a Help push button
- Selecting an item from a Help submenu.

Query and Set Object-Specific Help

To query and set object-specific help for an instance of the object override wpQueryDefaultHelp and wpSetDefaultHelp.

The wpQueryDefaultHelp method queries the default help panel ID and help library for this instance of the object class, as shown in the following figure: It automatically returns the class default help, if no object-specific help has been provided.

```
wpQueryDefaultHelp (WPObject *somSelf,
                    PULONG pHelpPanelId,
                    PSZ HelpLibrary)

where
       pHelpPanelId  - Pointer to the help panel ID
       HelpLibrary   - Help library name returned in the buffer
```

The wpSetDefaultHelp method sets the default help panel ID and help library for this instance of the object class, as shown in the following figure: It allows different instances to have different helps displayed. Using the value zero for HelpPanelId and NULL for HelpLibrary, will reset the object to the default class help panel.

```
wpSetDefaultHelp (WPObject *somSelf,
                  ULONG HelpPanelId,
                  PSZ HelpLibrary)

where
       HelpPanelId   - Help panel identity
       HelpLibrary   - Help library name returned in the buffer
```

Setup Strings

An object's default help panel ID and help library can be set using setup strings. The pszSetupString contains a series of keynames in which each is separated by semicolons, and which changes the behavior of the object. The following table shows the keynames used to set the default help panel ID and help library:

```
 Keyname                        Description

 HELPPANEL = id                 Sets the object's default help
                                panel. This is equivalent to
                                calling wpSetDefaultHelp.
```

```
 HELPLIBRARY = myobject.hlp     Sets the default help library
                                for the object's instance.
                                This is equivalent to calling
                                wpSetDefaultHelp.
```

The following methods can be called to set the object's default help panel ID and help library using setup strings:

- wpSetup(WPObject *self, PSZ pszSetupString)
- WinCreateObject(PSZ pszClassName, PSZ pszTitle, PSZ pszSetupString, PSZ pszLocation, ULONG ulFlags)
- WinSetObjectData(HOBJECT hObject, PSZ pszSetupString)

REXX Functions

The following REXX functions can be called to set the help panel ID and help library:

- SysCreateObject(classname, title, location, setup, Flag)
- SysSetObjectData(objectid, setup).

Using Help Methods

This section explains, through the use of sample code fragments, how to:

- Display the Help dialog for a pop-up menu item
- Modify the Help dialog for a pop-up menu item
- Query when a user requests help from an application window.

Displaying the Help Dialog for a Pop-Up Menu Item

To display the Help dialog for a pop-up menu item, override wpMenuItemHelpSelected. Derived classes can override this method to:

- Add the Help dialog for a new pop-up menu item
- Modify the Help dialog for an existing menu item.

The three following sample codes show how to override wpMenuItemHelpSelected. They present the class interface definition (IDL), source code (C), and Information Presentation Facility (IPF) files, respectively.

```
passthru: C.ih;
   .
   .
   #define IDM_CLOSEVIEWS       (WPMENUID_USER+1)
   #define ID_HELP_CLOSEVIEWS   257
   .
   .
endpassthru;

//# Method modifiers
   .
   .
wpMenuItemHelpSelected: override;
   .
   .
```

The following sample code shows the source code for MYOBJECT when using wpMenuItemHelpSelected:

```
/*
 *  METHOD: wpMenuItemHelpSelected
 *
 *  DESCRIPTION:
 *
 *    Process the help submenu item that was added.
 */
SOM_Scope BOOL SOMLINK myobj_wpMenuItemHelpSelected(MYOBJECT *somSelf,
                                                    ULONG MenuId)

{
  MYOBJECTData *somThis = MYOBJECTGetData(somSelf);
  MYOBJECTMethodDebug("MYOBJECT","myobj_wpMenuItemHelpSelected");

  switch(MenuId)
  {
    case IDM_CLOSEVIEWS:

    /* Append the new help library onto the system library, by calling
     * wpDisplayHelp(WPObject *self, ULONG HelpPanelId, PSZ HelpLibrary).
     * wpDisplayhelp sends a message to the help manager to display the help
     * panel.  ID_HELP_CLOSEVIEWS should have the same value as the resid in
     * the Information Presentation Facilty (MYOBJECT.IPF), that is, in this
     * particular case, ID_HELP_CLOSEVIEWS is defined as 257 and the resid
     * for the context menu help item in MYOBJECT.IPF is also 257.
     */

    return(_wpDisplayHelp(somSelf,ID_HELP_CLOSEVIEWS,szHelpLibrary));
    break;

    default:
    break;
  }

  /* Always return the parent's method in case of default */
  return (parent_wpMenuItemHelpSelected(somSelf,MenuId));
}
```

The following sample code illustrates the OS/2 Information Presentation Facility (IPF) for the class MYOBJECT:

```
.*================================================================*\
.*                                                                *
.* myobject.ipf - Information Tag Language file for the MYOBJECT *
.*                Class sample application's help manager         *
.*                                                                *
.*================================================================*/
:userdoc.

.*----------------------------------------------------------------*\
.*  Default Help for MYOBJECT class                               *
.*      res = ID_HELP_DEFAULT                                     *
.*----------------------------------------------------------------*/
:h1 res=256 name=ID_HELP_DEFAULT.MYOBJECT Class -- Help
:i1 id=aboutMain.About MYOBJECT Class
:p.This Class is a subclass of WPObject.

.*----------------------------------------------------------------*\
.*  Context Menu Item help                                        *
.*      res = ID_HELP_CLOSEVIEWS                                  *
.*----------------------------------------------------------------*/
:h1 res=257 name=ID_HELP_CLOSEVIEWS.Close Views Menu Item
:i1 id=IV.CloseViews
:p.When this context menu item is chosen, all currently open views
of this folder will be closed, except for the view that was
initially opened.  If no views are currently open, then no action will
be taken.

:euserdoc.
```

Requesting Help from an Application Window

When a user requests help from an application window by pressing the F1 key, the wpclsQueryDefaultHelp method should be overridden, as shown in the following figure:

```
wpclsQueryDefaultHelp(M_MYOBJECT *somSelf,
                      PULONG pHelpPanelId,
                      PSZ HelpLibrary)

where
       pHelpPanelId  - Pointer to the help panel ID
       HelpLibrary   - Help library name returned in the buffer
```

This method is called to allow the class object to specify its default help panel for its instances. This class method is called during the default processing of wpQueryDefaultHelp. It can be called at any time in order to determine the default help panel for the object class. The WPObject class does not process this method other than returning FALSE. Derived classes can override this method to provide the default help panel ID and help module name for their object instances. All classes should override this method to tell the system what help they want displayed when the user selects the F1 key. In this case, also, the value of ID_HELP_DEFAULT should be the same as the resid for the default help for MYOBJECT class in the IPF file. The three following sample codes show how to override wpclsQueryDefaultHelp. They present the class interface definition (IDL), source code (C), and make (MAK) files, respectively.

```
passthru: C.ih;
    .
    .
  #define ID_HELP_DEFAULT  256
    .
    .

endpassthru;

//# Class modifiers
    .
    .
wpclsQueryDefaultHelp: override;
    .
    .
```

The following sample code shows the source code of MYOBJECT when overriding wpclsQueryDefaultHelp:

```
/*
 *  METHOD: wpclsQueryDefaultHelp
 *
 *  DESCRIPTION:
 *
 *    Process input from the help submenu item that was added.
 */
SOM_Scope BOOL SOMLINK myobjM_wpclsQueryDefaultHelp(M_MYOBJECT *somSelf,
                                                    PULONG pHelpPanelId,
                                                    PSZ pszHelpLibrary)

{
  /* M_MYOBJECTData *somThis = M_MYOBJECTGetData(somSelf); */
  M_MYOBJECTMethodDebug("M_MYOBJECT","myobjM_wpclsQueryDefaultHelp");

  if (pHelpPanelId)    /* Set the default help panel ID */
    *pHelpPanelId = ID_HELP_DEFAULT;

  if (pszHelpLibrary)  /* Copy the help file name */
    strcpy(pszHelpLibrary, szHelpLibrary);

  return (TRUE);
}
```

The help library can be created by using the IPFC compiler. The following figure shows the make file of MYOBJECT:

```
myobject.hlp:  $$(@B).ipf
```

```
ipfc $*.ipf
```

Object Containers: Container Methods

A *container* is a visual component that holds objects. CUA container control is the primary way of interacting with Workplace Shell objects. All objects with which the user interacts are simply records that have been inserted into a container control. Any Workplace Shell object can be inserted into any container control created on the Workplace process. This chapter describes the different views available for containers, and the methods used to create and manipulate containers in Workplace Shell applications.

About Container Methods

A container can display objects in various formats and views. Generally speaking, each view displays different information about each object. If a container's data is too large for the window's work area, scrolling mechanisms are enabled. Containers are an integral component of the CUA user interface.

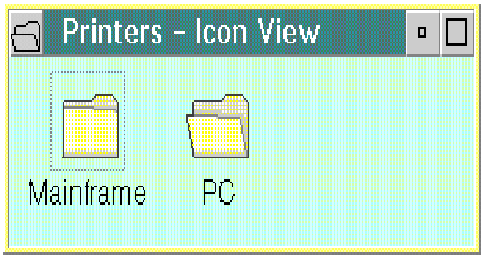
Container Views

A *Container view* lists the components of an object. The components can be ordered or unordered in the view; the order of the information displayed in a contents view does not affect the meaning of the object containing the information. CUA guidelines describe three kinds of Container views:

- Icon view

 An *Icon view* displays each object as an icon. Its purpose is to give the user an easy way to change the position of objects or to otherwise directly manipulate them.

 An object usually is represented by only one icon. However, for some tasks, the user might find it convenient to represent an object with more than one icon. For example, the user might want a representation of a printer object in more than one place so that the user can have easy access to the printer from anywhere. The user can create an additional icon, known as a *shadow* , to represent the same printer object. The following figure shows an Icon view with objects:

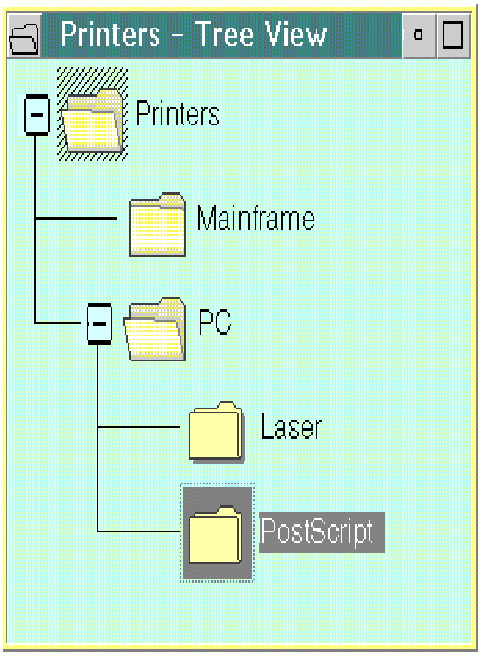
- Tree view

 The *Tree view* displays container objects arranged hierarchically.

 The leftmost objects displayed in the Tree view are at the *root level* and are the same objects displayed in all the other Container views. Objects that contain other objects are called *parent objects* . The objects that a parent object contains are called *child objects* and can be displayed only in the Tree view. Child objects that contain other objects serve a dual role: they are the children of their parent object, but they are parent objects as well, with children of their own. For example, a parent object might be a book that contains individual child objects for its chapters or a folder that contains several reports. The chapters or reports, in turn, could be parent objects that contain their own children, such as the major sections of a chapter or report.

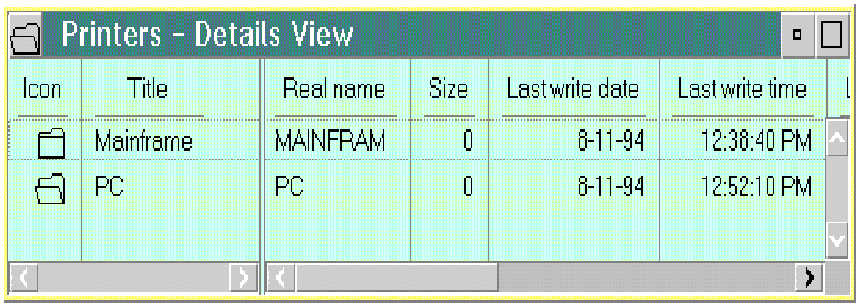
 If the child objects of a parent object are not displayed, the parent object can be *Expanded* to display them as a new branch in the Tree view. Once a parent item has been expanded, it can be *Collapsed* to remove its child objects from the display. The following

figure shows a Tree view with objects:



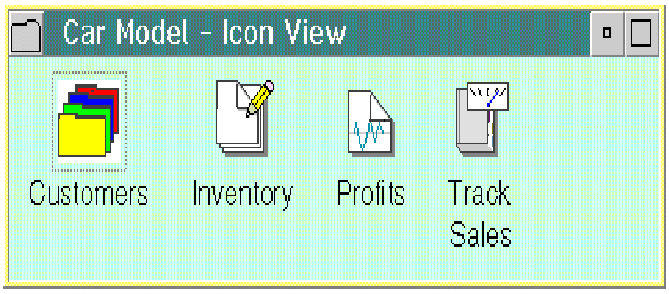
- Details view

 A *Details view* combines small icons with text that provides additional information about objects. The type of information displayed depends on the type of object and the type of tasks the user wants to perform. A Details view gives the user access to some of the object's more frequently used information, without requiring the user to open the object. Small icons are included in a Details view to provide a way for the user to easily recognize objects and to directly manipulate each object. Some Workplace Shell classes define a set of information that a user can display in a Details view of all instances of objects belonging to the class. The following figure shows a Details view with objects:



Container Objects

A *container object* holds other objects. Its primary purpose is to provide a way for the user to group related objects for easy access and retrieval. To illustrate its importance, consider the example of the design of a software model of a car dealership. A salesperson needs a car object to represent each car model on his lot. He also needs a customer object to represent each customer who purchases a car. He needs a worksheet object to track sales, profits, inventory, customers, and so on. Finally, he needs a container objects to group these objects. The following figure shows a container holding several related objects:



Using Container Methods

This section provides an example of what the Workplace Shell covers versus the container control. Secondly, it explains why the Workplace Shell subclasses the owner of a container and the frame window. Finally, this section describes the Workplace Shell container methods. More specifically, it explains how to:

- Insert one object into a container
- Insert multiple objects into a container
- Remove an object from a container
- Set emphasis on an object
- Register a new open view
- Obtain the object's MINIRECORDCORE pointer
- Obtain a pointer to an object

Workplace Shell and the Container Control

CUA container control is the primary key of interacting with Workplace Shell objects. All objects with which the user interacts are simply records that have been inserted into a container control.

The container control uses RECORDCORE and MINIRECORDCORE structures to represent the items that it holds. Memory for these structures is allocated by sending the container a CM_ALLOCRECORD message. When memory is allocated, the records may be inserted using the CM_INSERTRECORD message. The Workplace Shell provides object instance methods that perform all of this for you.

When an object is created, the Workplace Shell automatically allocates memory for the MINIRECORDCORE structure and stores the pointer in the object's instance data. Only one MINIRECORDCORE per object is used even if the object exists in multiple views.

The wpQueryCoreRecord method can be used to obtain the pointer to the MINIRECORDCORE structure that was allocated when the object was created. You would use this method to obtain a pointer to the MINIRECORDCORE structure that the container is using to represent the object. This may be necessary if you need to perform a task and no instance method for that task exists. If the container can perform the task, the application can use _wpQueryCoreRecord to obtain the PMINIRECORDCORE and send the container a message explicitly. The OBJECT_FROM_PREC macro performs the inverse function of _wpQueryCoreRecord and is used to obtain a pointer to the object (self) when the pointer to the MINIRECORDCORE structure is known.

Workplace Shell Container Owner Subclass Procedure

Any Workplace Shell object can be inserted into any container control created on the Workplace process using wpCnrInsertObject. When an object is inserted into a container via _wpCnrInsertObject, the Workplace Shell subclasses the owner of the container, and intercept some messages and notifications from the container. In this way, the Workplace Shell provides support that maps container records to their WPObject counterparts, and related behaviors. In other words, using the object's instance methods to insert an object into a container allows the object to inherit all of the Workplace Shell features, such as context menu, and Pickup and Drop operation.

Note: Since the Workplace Shell subclasses the container control's owner, you can only have ONE container control into which you are inserting objects per owner. This means you may have to expand your window hierarchy.

When a view is registered, the Workplace Shell subclasses the frame window. The Workplace Shell adds the viewed object's context menu to the system menu of the view. The Workplace Shell assumes that the container window has an ID of FID_CLIENT and that the container's owner is a frame window. This means that the container window is the client area of a frame window that was created with WinCreateStdWindow.

When a view of an object is opened, a USAGE_OPENVIEW item is added to the object's in-use list. The USAGE_OPENVIEW items are used to automatically switch to the open view when the user chooses to open the same view again. This behavior is controlled by wpOpen or wpSwitchTo. For more detail on the USAGE_OPENVIEW item, see Object Usage: Usage Methods.

Inserting Container Objects

The wpCnrInsertObject and wpclsInsertMultipleObjects methods are used to insert objects into a container window. Objects inserted using these methods automatically inherit the default Workplace Shell behavior such as context menus, and Pickup and Drop capability. When an object is inserted into a container, the Workplace Shell will add a USAGE_RECORD item to the object's in-use list. There will be one of these USAGE_RECORD items for each view (container window) that the object is inserted in. These methods send the CM_INSERTRECORD message to the container window.

Inserting One Object

The wpCnrInsertObject method inserts a record into a container control window. The following shows the syntax of wpCnrInsertObject:

```
#define INCL_WINWORKPLACE
#include <os2.h>

PMINIRECORDCORE wpCnrInsertObject (WPObject *self,
                                   HWND hwndCnr,
                                   PPOINTL pIcon,
                                   PMINIRECORDCORE pParent,
                                   PRECORDINSERT pRecInsert)
```

The *self* (WPObject *) parameter is the pointer to the object on which the method is being invoked. It points to an object of class WPObject.

The *hwndCnr* (HWND) parameter is the handle of the container control window.

The *pIcon* (PPOINTL) parameter is the pointer to a POINTL structure specifying the initial icon position in the container control window.

The *pParent* (PMINIRECORDCORE) parameter is the pointer to the parent record. The *pParent* parameter specifies the record of the immediate parent of the record being inserted. This parameter should be set to NULL, if the record has no parent or if the Tree view is not supported.

The *pRecInsert* (PRECORDINSERT) parameter is the pointer to the RECORDINSERT structure specifying how this record is to be inserted relative to other records in the same container. The possible values of this parameter are:

Value	**Description**
NULL	Insert the record into the next available position.
Other	Insert the record into the position specified by *pRecInsert*.

The *rc* (PMINIRECORDCORE) parameter is returned with one of the following values:

Value	**Description**
NULL	Error occurred.
Other	Pointer to the inserted record.

This method is used to give Workplace Shell object behavior (such as context menu support) to records inserted directly into a WC_CONTAINER control window. To remove records from the container, a call to wpCnrRemoveObject should be made. The following figure

shows the insertion of an object into a container control window:

```
/*************************************************************************/
/* This sample code assumes that the object has already been created    */
/* using wpclsNew and object's pointer has been obtained                */
/* using wpclsQueryObject.                                              */
/*************************************************************************/

WPObject *self;                         /* The object to be inserted  */
HWND hwndCnr;                           /* Handle of the container    */
                                        /* window                     */
POINTL ptlIcon;                         /* Location within container  */
                                        /* to place record            */
RECORDINSERT RecordInsert;              /* Container RECORDINSERT     */
                                        /* data structure             */

/*************************************************************************/
/* Set up the RECORDINSERT data structure.                              */
/* The object will be inserted at the beginning of the list of records. */
/*************************************************************************/

RecordInsert.cb = sizeof(RECORDINSERT);   /* Size of the RECORDINSERT   */
                                          /* data structure             */
RecordInsert.pRecordParent = NULL;        /* Insert record at root      */
                                          /* level                      */
RecordInsert.zOrder = CMA_TOP;            /* Top of the z-order         */
RecordInsert.cRecordsInsert = 1;          /* Inserting 1 record         */
RecordInsert.fInvalidateRecord = TRUE;    /* Invalidate record at the   */
                                          /* time it is inserted        */
RecordInsert.pRecordOrder =               /* Place record at beginning  */
  (PRECORDCORE)CMA_FIRST;                 /* of the list of records     */
                                          /* already inserted           */

/*************************************************************************/
/* Initialize the location where the object will be inserted.           */
/*************************************************************************/

ptlIcon.x = 50;
ptlIcon.y = 50;

/*************************************************************************/
/* Inserting object at location 50,50.                                  */
/*************************************************************************/

_wpCnrInsertObject(self, hwndCnr, &ptlIcon, NULL, &RecordInsert)
```

Inserting Multiple Objects

The wpclsInsertMultipleObjects method is used to insert multiple objects into a container window at one time. The following shows the syntax of wpclsInsertMultipleObjects:

```
#define INCL_WINWORKPLACE
#include <os2.h>

BOOL wpclsInsertMultipleObjects(M_WPObject *self,
                                HWND hwndCnr,
                                PPOINTL pptlIcon,
                                PVOID *pObjectArray,
                                PVOID pRecordParent,
                                ULONG NumRecords)
```

The *self* (M_WPObject *) parameter is the pointer to the WPObject class object.

The *hwndCnr* (HWND) parameter is the handle of the container window to insert objects into.

The *pptlIcon* (PPOINTL) parameter is the pointer to a POINTL structure specifying the coordinates for the position of the first object inserted.

The *pObjectArray* (PVOID *) parameter is the pointer to an array of object pointers. The objects in this array are to be inserted into the

container.

The *pRecordParent* (PVOID) parameter is the parent record pointer. This parameter should be set to NULL if the records being inserted will not have a parent or if the Tree view is not supported.

The *NumRecords* (ULONG) parameter is the number of objects in the *pObjectArray* parameter.

The *rc* (BOOL) parameter is returned with one of the following values:

Boolean	Description
TRUE	Objects successfully inserted into the container.
FALSE	Error occurred.

This method is used for the rapid insertion of multiple objects into the container all at once. The wpCnrInsertObject method performs the same function as this method but operates on only one object at a time. This method will add each of the objects to the in-use list. The following figure shows the insertion of multiple objects into a container control window:

```
BOOL wpclsInsertMultipleObjects(M_WPObject *self,
                                HWND hwndCnr,
                                PPOINTL pptlIcon,
                                PVOID *pObjectArray,
                                PVOID pRecordParent,
                                ULONG NumRecords)

M_WPObject *self;
HWND hwndCnr;
POINTL ptlIcon;
PVOID pObjectArray{5};
ULONG i;

/*************************************************************************/
/* pObjectArray has been filled in with pointers to objects              */
/* created using WinCreateObject or wpClsNew                             */
/*************************************************************************/

/*************************************************************************/
/* Initial location for first object being inserted.                     */
/*************************************************************************/

ptlIcon.x = 50;
ptlIcon.y = 50;

/*************************************************************************/
/* Inserting multiple objects.                                           */
/*************************************************************************/

_wpclsInsertMultipleObjects(self, hwndCnr, &ptlIcon, &pObjectArray, NULL,5);
```

Removing Container Objects

The wpCnrRemoveObject method should be called to remove an object from a container, if that object was inserted using either wpCnrInsertObject or wpclsInsertMultipleObjects. When the object is removed from a container, the USAGE_RECORD item, which was added to the in-use list when the object was inserted, will be removed. This method sends the CM_REMOVERECORD message to the container. The following shows the syntax of wpCnrRemoveObject:

```
#define INCL_WINWORKPLACE
#include <os2.h>

BOOL wpCnrRemoveObject (WPObject *self,
                        HWND hwndCnr,
                        PMINIRECORDCORE pRecord)
```

The *self* (WPObject *) parameter is the pointer to the object on which the method is being invoked. It points to an object of class WPObject.

The *hwndCnr* (HWND) parameter is the handle of the container control window.

The *pRecord* (PMINIRECORDCORE) parameter is the pointer to the record to be removed.

rc (BOOL) is returned with one of the following values:

Boolean	Description
TRUE	Record successfully removed.
FALSE	Error occurred.

The following figure shows the deletion of an object from a container control window:

```
/**************************************************************************/
/* This sample code assumes that the object has already been              */
/* created using wpclsNew and object's point has been                     */
/* obtained using wpclsQueryObject.                                       */
/**************************************************************************/

WPObject *self;              /* The object to be removed                  */
HWND hwndCnr;                /* Handle of the container window            */
PMINIRECORDCORE pRecord;     /* Pointer to container record               */

/**************************************************************************/
/* Removing the record from the container window.                         */
/**************************************************************************/

_wpCnrRemoveObject(self, hwndCnr, pRecord);
```

Setting Emphasis on Objects

The wpCnrSetEmphasis method provides a way to change the visual emphasis of an object. Visual emphasis can be used to give the user feedback as to the state of the object. For example, when a folder is opened on the Desktop, in-use emphasis is added to the folder icon. The Workplace Shell accomplishes this by calling wpCnrSetEmphasis on the object.

This method sends the CM_SETRECORDEMPHASIS message to the container. The CRA_* flags that are specified for the *ulEmphasisAttr* parameter are the same as those for the CM_SETRECORDEMPHASIS container message. The following shows the syntax of wpCnrSetEmphasis:

```
#define INCL_WINWORKPLACE
#include <os2.h>

BOOL wpCnrSetEmphasis (WPObject *self, ULONG ulEmphasisAttr, BOOL fTurnOn)
```

The *self* (WPObject *) parameter is the pointer to the object on which the method is being invoked. It points to an object of class WPObject.

The *ulEmphasisAttr* (ULONG) parameter is the type of emphasis to set on the object. The CRA_* flags are as follows. These flags are the container's record attribute flags. The flags may be combined using the logical OR (|) operator.

Emphasis Type	Description
CRA_CURSORED	Specifies that the record will be drawn with a selection cursor.
CRA_DISABLED	Specifies that the record will be drawn with unavailable-state emphasis.
CRA_INUSE	Specifies that a record will be drawn with in-use emphasis.
CRA_PICKED	Specifies that the record will be drawn with "picked" emphasis. It is part of the Pickup and Drop operation.
CRA_SELECTED	Specifies that a record will be drawn with selected-state emphasis.
CRA_SOURCE	Specifies that the record will be drawn with source-menu emphasis.

The *fTurnOn* (BOOL) parameter sets or resets the specified attribute.

Boolean	Description
TRUE	Set the specified attribute.
FALSE	Reset the specified attribute.

rc (BOOL) is returned with one of the following values:

Boolean	Description
TRUE	Attribute successfully set.
FALSE	Error occurred.

The following figure shows how to set the emphasis on a container object:

```
/*************************************************************************/
/* This sample code assumes that the object has already been created    */
/* using wpclsNew and object's pointer has been obtained                */
/* using wpclsQueryObject.                                              */
/*************************************************************************/

WPObject *self;              /* The object whose emphasis is being set  */
ULONG ulEmphasis;            /* ULONG data structure                    */

ulEmphasis = CRA_INUSE;      /* Emphasis type desired                   */

/*************************************************************************/
/* Setting emphasis on the object.                                      */
/*************************************************************************/

_wpCnrSetEmphasis(self, ulEmphasis, TRUE);
```

Registering a New Open View

The wpRegisterView method should be called whenever a new view of an object is created. Registering a view will set the title of the frame window to the object's title and add the view title as the current view in the window list. The wpRegisterView method is used to allow an object to register a new open view. The following shows the syntax of wpRegisterView:

```
#define INCL_WINWORKPLACE
#include <os2.h>

BOOL wpRegisterView (WPObject *self, HWND hwndFrame, PSZ pszViewTitle)
```

The *self* (WPObject *) parameter is the pointer to the object on which the method is being invoked. It points to an object of class WPObject.

The *hwndFrame* (HWND) parameter is the handle of the frame window containing the new view.

The *pszViewTitle* (PSZ) parameter is the pointer to a string containing the name of the view.

The *rc* (BOOL) parameter is returned with one of the following values:

Boolean	Description
TRUE	New view successfully registered.
FALSE	Error occurred.

Registering a view will set the object title as the title of the frame window and add a view title as the current view in the window list and titlebar. The following figure shows how to register a new open view of an object:

```
WPObject *self;
HWND hwndFrame;
PSZ pszViewTitle="OS/2 Programs - Icon View";
```

```
_wpRegisterView(self, hwndFrame, pszViewTitle);
```

Obtaining the Object's MINIRECORDCORE Pointer

The wpQueryCoreRecord method is used to obtain a pointer to the MINIRECORDCORE structure associated with a given object. When an object is created, the Workplace Shell automatically allocates memory for a MINIRECORDCORE structure and stores a pointer to it in the object's instance data. The following shows the syntax of wpQueryCoreRecord:

```
#define INCL_WINWORKPLACE
#include <os2.h>

PMINIRECORDCORE wpQueryCoreRecord (WPObject *self);
```

The *self* (WPObject *) paramter is the pointer to the object on which the method is being invoked. It points to an object of class WPObject.

The *pRecord* (PMINIRECORDCORE) parameter is returned. It points to the MINIRECORDCORE structure for the object.

Every object has a MINIRECORDCORE structure associated with it so it can be placed into a container. This method is used to obtain the PMINIRECORDCORE when the pointer to the object is known. The following figure shows how to obtain a pointer to the MINIRECORDCORE structure associated with an object:

```
PMINIRECORDCORE pRecord;
WPObject *self;

pRecord = _wpQueryCoreRecord(self);
```

Obtaining a Pointer to an Object

The OBJECT_FROM_PREC macro is used to obtain a pointer to an object when the PMINIRECORDCORE is known. The following figure shows how to obtain a pointer to an object:

```
WPObject *self;
PMINIRECORDCORE pRecord;

self = OBJECT_FROM_PREC(pRecord);
```

Object Criteria: Details Methods

The class defines the object's instance data and methods. *Details data* are part of instance data and define more accurately the behavior of the object. This chapter describes the general concepts of Details data and explains how to use the details methods in Workplace Shell applications.

About Details Methods

The basic unit of organization of SOM programs is the object. SOM objects consist of instance data and methods. SOM classes define the behavior of sets of like objects. You can think of a SOM class as a description of an object. The class defines the object's instance data and methods. *Details data* are part of instance data and define more accurately the behavior of the object.

Not all classes have the Details data section defined. Only those classes that have a requirement to store such information define this section as part of the instance data. After the Details data are defined, it is the class creator's responsibility to set and use them correctly. Details data allows you to:

- Define, for a class of objects, some confidential data along with general data
- Have a special container Details view
- Choose specific Details fields to be displayed in Details view by filtering out unwanted fields
- Filter objects that match or do not match a set of criteria
- Search for objects based on a single or complex set of criteria
- Sort objects.

The Details data of all objects contained in a container can be displayed in a Details view. The fields that are displayed in the Details view correspond to the Object type (class object) that is selected in the View page (page three of the View pages) of the Settings notebook. Once the Object type selected, the fields are listed in the Details to display drop-down list box. the following figure shows an example of the Details view for a Workplace Shell class. Each record provides information for a specific instance of an object belonging to the class.

```
                    Details for Instances of MyObject Class

             Icon     Title    Object Class    Field1   Field2   Field3

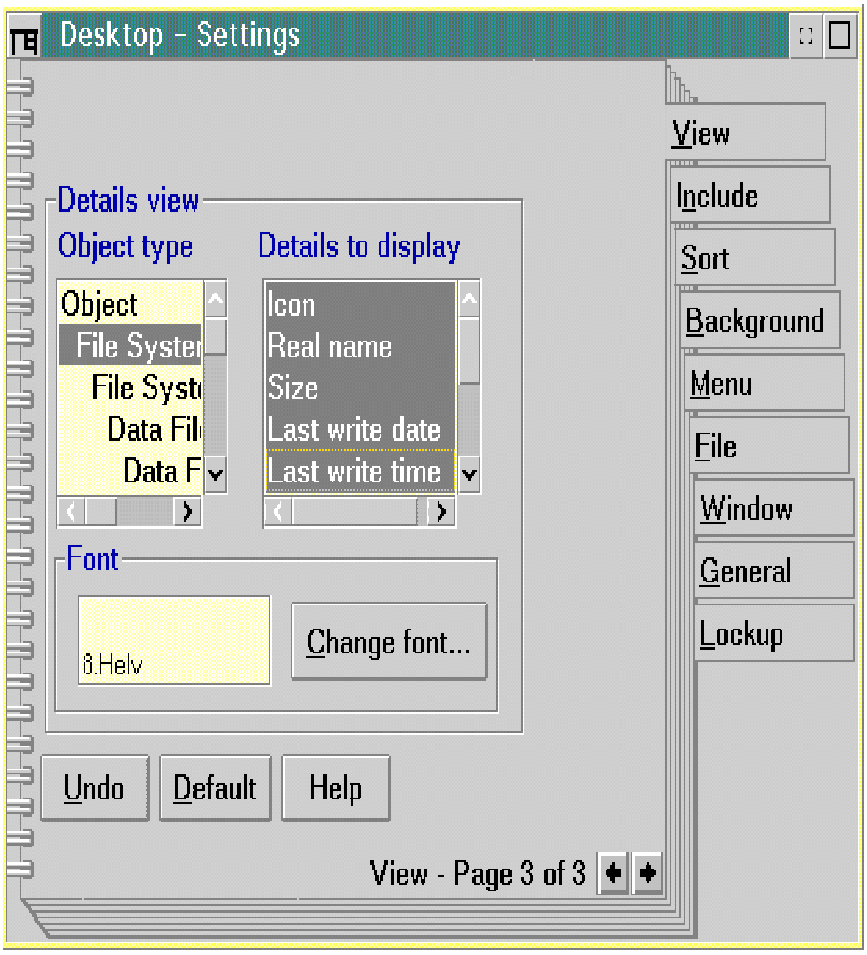
  Instance 1

  Instance 2
```

This Details view window is two windows separated by a split bar. The left side of the window contains, at least, the Title column (the default column) and can also contain the Details fields available for all classes and, in turn, for all objects. The right side of the window can be empty, if the objects to be represented are not descended from the chosen class. The user can choose:

- Which class provides Details fields to the window
- Which Details fields of this class are visible.

By default, the system sets the WPFileSystem class that provides all Details fields to be visible. The Workplace Shell provides a Settings notebook, page three of the View pages, to set the class and Details fields to be displayed in the Details view. The following figure shows the Settings notebook, page three of the View pages:



A record that contains information for an object instance of the class is created by overriding wpQueryDetailsData. The column headings for a Details view are specified by overriding wpclsQueryDetailsInfo.

An object can inherit the set of Details data defined by its ancestors. A record containing Details data for an object can also contain sets of Details data from objects that are defined by its parent or ancestors. The Workplace Shell handles this by creating records of contiguous blocks of memory. To inherit the details defined by ancestor classes, wpQueryDetailsData calls the parent class wpQueryDetailsData, which calls its parent class wpQueryDetailsData, and so forth, until the oldest ancestor that defined Details data is reached. In addition to inheriting Details data from its ancestors, the object can have some of its own. For each call to wpQueryDetailsData, a variable-length block of memory containing a set of Details data for the object is added to the record. A pointer is moved to the end of the last block of memory added to the record, where the next call to wpQueryDetailsData adds the next block of memory. Each block in the record contains a set of Details data for the object defined by its class or by one of its ancestor classes, as shown in the following figure:

```
Details Defined   Details Defined   Details Defined
by MyObject's     by MyObject's     by MyObject
Ancestor          Parent
```

The format for Details data contained in a record is defined by overriding wpclsQueryDetailsInfo. The format is defined in a linked list of CLASSFIELDINFO data structures for each Details field in a record. This linked list is created the same way as a record: wpclsQueryDetailsInfo calls its parent class wpclsQueryDetailsInfo, which calls its parent class wpclsQueryDetailsInfo, and so forth. Each call adds a set of CLASSFIELDINFO data structures to the linked list until the list contains all CLASSFIELDINFO data structures for each Details field of the object's Details record, as shown in the following figure:

```
  Format of          Format of                  Format of
  Details Data       Details Data               Details Data
  Defined by         Defined by                 Defined by
  its Ancestor       its Parent                 MyObject

>Field1 >Field2    >Field3 >Field4    >Field5 >Field6 >Field7 >Field8   >
```

The CLASSFIELDINFO data structure describes the attributes of Details data for each field.

Using Details Methods

The CAR sample Workplace Shell object in the Toolkit can be used to illustrate the requirements for defining Details data for a class of objects. CAR can provide Details data for CAR objects by:

1. Defining a data structure for the Details fields to be included in the Details view, as shown in the following sample code:

```
/* CARDETAILS:  Structure used for Details view  */

typedef struct _CARDETAILS
{
   PSZ      pszMake;     /* Manufacturer     */
   PSZ      pszModel;    /* Model            */
   PSZ      pszColor;    /* Color of car     */
   CDATE    cdateSale;   /* Date of sale     */
   ULONG    ulPrice;     /* Price in dollars */
}
CARDETAILS;
typedef CARDETAILS *PCARDETAILS;
```

2. Defining and initializing a static array of CLASSFIELDINFO structures for each set of Details data, as shown in the following sample code. This is done on class initialization, when wpclsInitData is called.

 Note: Title and Icon details are defined for CAR objects by its ancestor class WPObject. This means that the CLASSFIELDINFO data structures for Title and Icon details are defined and initialized in the WPObject class definition.

```
#define NUM_CAR_FIELDS 5

CLASSFIELDINFO fieldinfo[NUM_CAR_FIELDS];

PSZ pszCarColTitles[] =
{
   "Make",           /* Details column 1 */
   "Model",          /* Details column 2 */
   "Color",          /* Details column 3 */
   "Sale date",      /* Details column 4 */
   "Price ($)"       /* Details column 5 */
};
    .
    .
    .

/************************************************************/
/* Method:  wpclsInitData                                   */
/* Purpose: Initialize the Details data                     */
/************************************************************/

SOM_Scope void SOMLINK carM_wpclsInitData(M_Car *somSelf)

{
  USHORT          rc, i;
  PCLASSFIELDINFO pCFI;

  /* M_CarData *somThis  = M_CarGetData(somSelf); */
  M_CarMethodDebug("M_Car","carM_wpclsInitData");

  /* Call the parent class method first */
  parent_wpclsInitData(somSelf);

  /*   Get the class title */
  if (!WinLoadString(WinQueryAnchorBlock(HWND_DESKTOP),
                     _clsQueryModuleHandle(somSelf),
                     ID_TITLE,
                     sizeof(szCarClassTitle), szCarClassTitle))
    /* If the load string has failed, use the parent's string */
    strcpy(szCarClassTitle, parent_wpclsQueryTitle(somSelf));

  /* Initialize everything needed for the CLASSFIELDINFO */
  /* data structure for the CAR object class             */
```

```
  for (i=0,pCFI=fieldinfo;i<NUM_CAR_FIELDS;i++,pCFI++)

  {
    /* Memset the structure to zero's */
    memset((PCH)pCFI,0,sizeof(CLASSFIELDINFO));

    pCFI->cb             = sizeof(CLASSFIELDINFO);

    pCFI->flData         = CFA_RIGHT     |
                           CFA_SEPARATOR |
                           CFA_FIREADONLY;

    pCFI->flTitle        = CFA_CENTER        |
                           CFA_SEPARATOR     |
                           CFA_HORZSEPARATOR |
                           CFA_STRING        |
                           CFA_FITITLEREADONLY;

    pCFI->pNextFieldInfo = pCFI + 1;   /* Point to the next CLASSFIELDINFO */
    pCFI->pTitleData     = (PVOID) apszCarColTitles[i];
    pCFI->flCompare      = COMPARE_SUPPORTED |
                           SORTBY_SUPPORTED;

    switch (i)
    {
      case INDEX_MAKE:
        pCFI->flData           |= CFA_STRING;
        pCFI->offFieldData      = (ULONG)(FIELDOFFSET(CARDETAILS,pszMake));
        pCFI->ulLenFieldData    = sizeof(PSZ);
        pCFI->DefaultComparison = CMP_EQUAL;
        break;

      case INDEX_MODEL:
        pCFI->flData           |= CFA_STRING;
        pCFI->offFieldData      = (ULONG)(FIELDOFFSET(CARDETAILS,pszModel));
        pCFI->ulLenFieldData    = sizeof(PSZ);
        pCFI->DefaultComparison = CMP_EQUAL;
        break;

      case INDEX_COLOR:
        pCFI->flData           |= CFA_STRING;
        pCFI->offFieldData      = (ULONG)(FIELDOFFSET(CARDETAILS,pszColor));
        pCFI->ulLenFieldData    = sizeof(PSZ);
        pCFI->DefaultComparison = CMP_EQUAL;
        break;

      case INDEX_SALE_DATE:
        pCFI->flData           |= CFA_DATE;
        pCFI->offFieldData      = (ULONG)(FIELDOFFSET(CARDETAILS,cdateSale));
        pCFI->ulLenFieldData    = sizeof(CDATE);
        pCFI->ulLenCompareValue = sizeof(CDATE);
        pCFI->DefaultComparison = CMP_GREATER;
        break;

      case INDEX_PRICE:
        pCFI->flData           |= CFA_ULONG;
        pCFI->offFieldData      = (ULONG)(FIELDOFFSET(CARDETAILS,ulPrice));
        pCFI->ulLenFieldData    = sizeof(ULONG);
        pCFI->ulLenCompareValue = sizeof(ULONG);
        pCFI->DefaultComparison = CMP_GREATER;
        break;

    }  /* End switch(i) */
  }   /* End for (i=0,pCFI=fieldinfo;i<NUM_CAR_FIELDS;i++,pCFI++) */
  /* Terminate linked list */
  fieldinfo[NUM_CAR_FIELDS-1].pNextFieldInfo = NULL;
}   /* End carM_ wpclsInitData() */
```

3. Defining the column headings and format of the Details data placed into the object's Details record by overriding wpclsQueryDetailsInfo, as shown in the following sample code:

```
/************************************************************/
/* Method: wpclsQueryDetailsInfo                            */
/* Appends the CAR specific chain of FIELDINFO data structure   */
/* describing the Details data of this object to            */
/* *ppClassFieldInfo, if ppClassFieldInfo is non-NULL.      */
/* In this case, it also sets *ppClasssFieldInfo to the head    */
/* of the linked list.                                      */
/*                                                          */
```

```
/* Adds the number of bytes required by the Details data for    */
/* car to *pSize, if pSize is non-NULL.                         */
/*                                                              */
/* Add Details data for this object.                            */
/****************************************************************/

SOM_Scope ULONG SOMLINK
  redcarM_wpclsQueryDetailsInfo(M_Car *somself,
                                PCLASSFIELDINFO *ppClassFieldInfo,
                                PULONG pSize)

{
  ULONG           cParentColumns;
  PCLASSFIELDINFO pCFI;
  ULONG           i;

  /* M_CarData  *somThis = M_CarGetData(somself); */
  M_CarMethodDebug("M_Car","carM_wpclsQueryDetailsInfo");

  /* Always call the parent method first to retrieve the number */
  /* of Details columns and parent's data already defined in    */
  /* the Details buffer.                                        */
  cParentColumns =
    parent_wpclsQueryDetailsInfo(somself,ppClassFieldInfo,pSize);

  /* If pSize is non-NULL, we must add the size of our */
  /* Details column data structure.                    */
  if (pSize)
    *pSize += sizeof(CARDETAILS);  /* Adjust size */

  /****************************************************************/
  /* If the request was for the chained FIELDINFO data structures    */
  /* (ppClassFieldInfo is non-NULL), link them in.                   */
  /* Eventually the chain will look like this:                       */
  /*                                                                 */
  /* Grandparent - Parent - Me - Child - Grandchild.                 */
  /*                                                                 */
  /* Get the pointer to the beginning of the chain.                  */
  /* If the beginning of the chain is 0, assign the address          */
  /* of the first CLASSFIELDINFO structure to *ppClassFieldInfo.     */
  /* Otherwise, *ppClassFieldInfo points to the first column         */
  /* description in the chain.  The chain is examined and            */
  /* the CLASSFIELDINFO data structure is linked to the end of it.   */
  /****************************************************************/

  if (ppClassFieldInfo)
  {
    /* Find the last link in the chain; then link the CLASSFIELDINFO */
    /* data structures to the chain.                                 */
    if (*ppClassFieldInfo)
    {
      pCFI = *ppClassFieldInfo;
      for (i=0;i<cParentColumns;i++)
        pCFI = (pCFI->pNextFieldInfo) ? pCFI->pNextFieldInfo : pCFI;

      pCFI->pNextFieldInfo = fieldinfo;
    }
    else
      *ppClassFieldInfo = fieldinfo;
  }
  return ((ULONG) (cParentColumns + NUM_CAR_FIELDS));
}   /* End carM_wpclsQueryDetailsInfo() */
```

4. Adding Details data to an object's Details record by overriding wpQueryDetailsData, as shown in the following sample code:

```
/*
 *  Returns the car specific data for the Details view of this object.
 *  Sets the pointer (*ppDetailsData) to the beginning of the buffer
 *  into which the data is written.
 */

SOM_Scope ULONG SOMLINK car_wpQueryDetailsData(Car *somSelf,
                                               PVOID *ppDetailsData,
                                               PULONG pcp)

{
  PCARDETAILS pCarDetails;
  PBYTE       pSize;
  CarData *somThis = CarGetData(somSelf);
```

```
  CarMethodDebug("Car","car_wpQueryDetailsData");

  parent_wpQueryDetailsData(somSelf,ppDetailsData,pcp);

  if (ppDetailsData)    /* Query data */
  {
    pCarDetails                 = (PCARDETAILS) *ppDetailsData;
    pCarDetails->pszMake        = "Toyota";   /* Manufacturer     */
    pCarDetails->pszModel       = "Camry";    /* Model name       */
    pCarDetails->pszColor       = "BLUE";     /* Color of the car */
    pCarDetails->cdateSale.day  = 24;         /* Date of sale     */
    pCarDetails->cdateSale.month = 12;
    pCarDetails->cdateSale.year  = 91;
    pCarDetails->ulPrice        = 14000;      /* Price in dollars */

    /* Point to buffer location after the Details data */
    *ppDetailsData = ((PBYTE) (*ppDetailsData)) + sizeof(*pCarDetails);

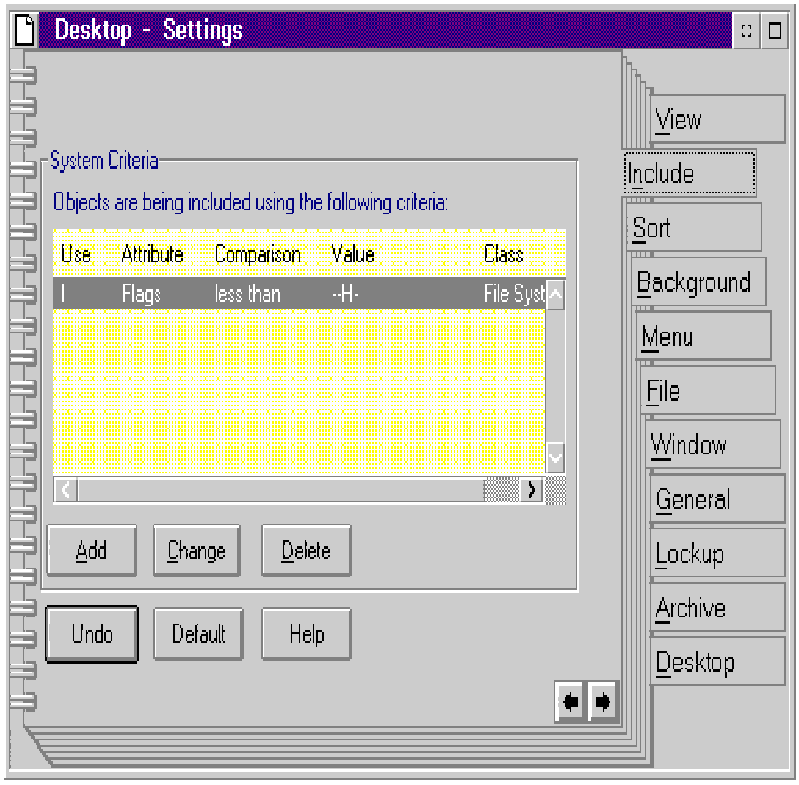
  }   /* End if (ppDetailsData) */
  else   /* Query size of data */
  {
    /* Caller is querying the buffer size */
    *pcp += sizeof(*pCarDetails);
  }
  return(TRUE);
}   /* End car_wpQueryDetailsData() */
```

Including Objects in a Container

A container represents the standard mode to organize objects you work with. It simply stores objects and allows them to interact with the user. Sometimes, however, a container can contain unnecessary objects. For this reason, the Workplace Shell provides a powerful tool that allows you to filter objects you want to include in or to exclude from the container. You specify objects to be included in a container by selecting the Include tab of the Settings notebook. The filter for this container is specified by the criteria that tells the system to include, in the container, all objects that match the criteria. The default criteria is shown in the following figure:

```
I Flags less than --H-
```

This criteria means that all objects that do not have the *hidden* attribute set will be included in the Open view of the container. The following figure shows the Include dialog, which is set with the default criteria:



The Include dialog provides the following push buttons to manipulate criteria:

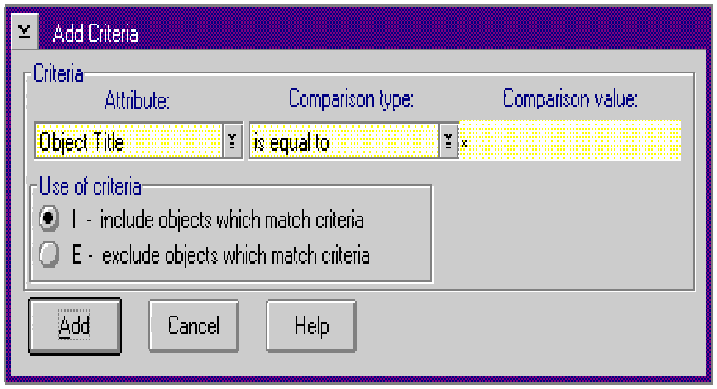
Push Button	**Description**
Add	Brings up a dialog for adding new criteria.
Change	Brings up a dialog for changing the selected criteria.
Delete	Deletes the selected criteria.
Undo	Resets the criteria to the state it was in when you opened the Include tab.
Default	Resets the criteria to the default criteria as shown in the second previous figure.
Help	Provides help on the Include tab.

As an example, to set the criteria to include nothing (exclude everything), delete the default criteria and add the criteria shown in the following figure:

```
E  ObjectClass  is equal to  Object
```

Adding Criteria

All objects in the system have properties such as name, date of creation, size, color, and so forth. A group of like objects make up a *class* . The objects in one class differ from the objects in another class by their properties. If a class is descended from another class, however, the objects from the descended class have, at least, all the properties of its parent, and can have some of their own. The following figure shows the Add Criteria dialog:



To create a valid criteria, four elements must be specified:

Element	Description
Attribute	Lists all the different attributes you can select for the criteria.
Comparison type	Lists all the different comparison types associated with the selected attribute.
Comparison value	Lists all the different values associated with the selected attribute.
Use of criteria	Contains two radio buttons: • To include objects that match the criteria. • To exclude objects that match the criteria.

The following table shows the attributes that are available in the system at installation time. The attributes are listed in the same order as the ones shown in the Attribute drop-down list box.

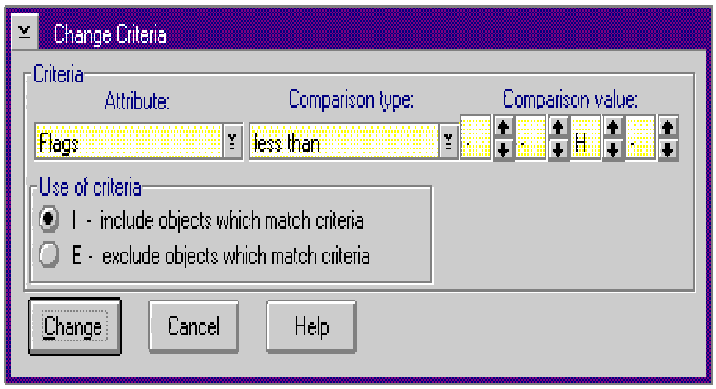
Attribute	Class
ObjectTitle	WPObject
ObjectStyle	WPObject
ObjectClass	WPObject
RealName	WPFileSystem
Size	WPFileSystem
LastWriteDate	WPFileSystem
LastWriteTime	WPFileSystem
LastAccessDate	WPFileSystem
LastAccesTime	WPFileSystem
CreationDate	WPFileSystem
CreationTime	WPFileSystem
Flags	WPFileSystem
ReadOnlyFlag	WPFileSystem
HiddenFlag	WPFileSystem
SystemFlag	WPFileSystem
DirectoryFlag	WPFileSystem
ArchivedFlag	WPFileSystem
Subject	WPFileSystem
Comments	WPFileSystem
Key Phases	WPFileSystem
History	WPFileSystem

```
Extended Attribute Size        WPFileSystem
```

After selecting the attribute, comparison type, and comparison value, select the appropriate radio button to specify whether to include or exclude objects that match the specified criteria.

Changing Criteria

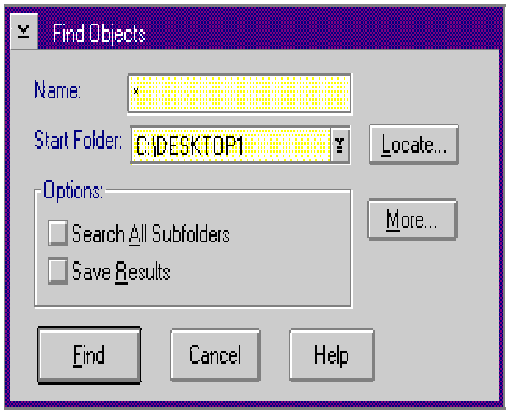
The following figure shows the Change Criteria dialog:



To change a criteria, specify the Attribute, Comparison type, Comparison value, or Use of criteria, and select the Change push button.

Searching for Objects

The Workplace Shell has a Find tool that allows you to search for objects based on several criteria. Every container pop-up menu in the system has a Find Objects dialog. The following figure shows the Find Objects dialog:



You use the Find Objects dialog to search for objects by their names, or by selecting the More push button and by specifying several criteria. The Find Objects dialog comes with default values:

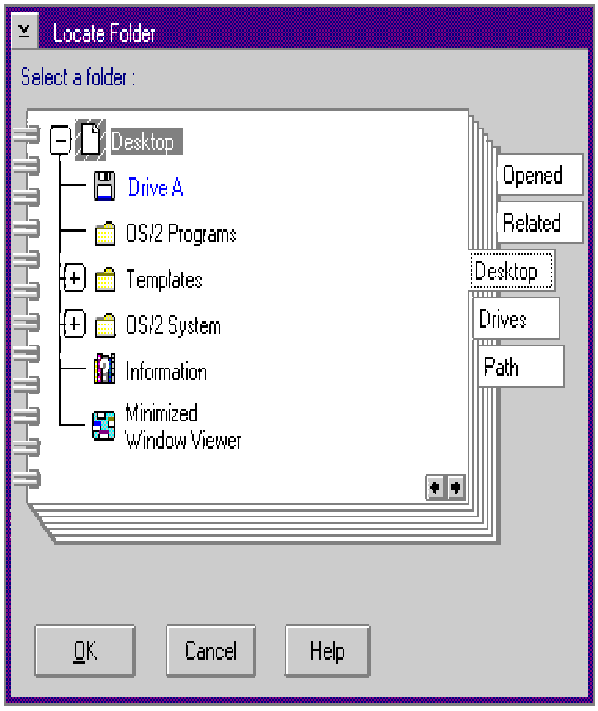
Default Values	**Description**
Name	Specifies the name of the object to search for. The asterisk (*) is the default and means to search for all object names.
Start Folder	Specifies where the search begins. The default is the path of the container from which the Find process is started. For the Desktop, the default is <ALL DRIVES>. The drop-down list box contains additional options, including the specific drives available in the system.
Search All Subfolders	Specifies to search all subfolders. The default is to search the specified folder.

Save Results

Creates a folder containing the Find result objects. The Find result objects are shadows of the objects, not the objects themselves. If this option is not checked, Find creates a window with icons that represent pointers to the real objects. This is the default option.

Locate

Allows you to specify another starting point for the Find process. The following figure shows the Locate dialog. The focus is on the Desktop tab.



The Locate dialog contains five options:

Option	Description
Desktop	Lists all folders in the Desktop folder.
Drives	Displays the contents of a drive in Tree view.
Opened	Lists all folders currently open on the Desktop.
Path	Specifies, in command line format, where the search begins. This is an alternative to Start Folder.
Related	Gives the Tree view for the drive from which the Find process started.

More

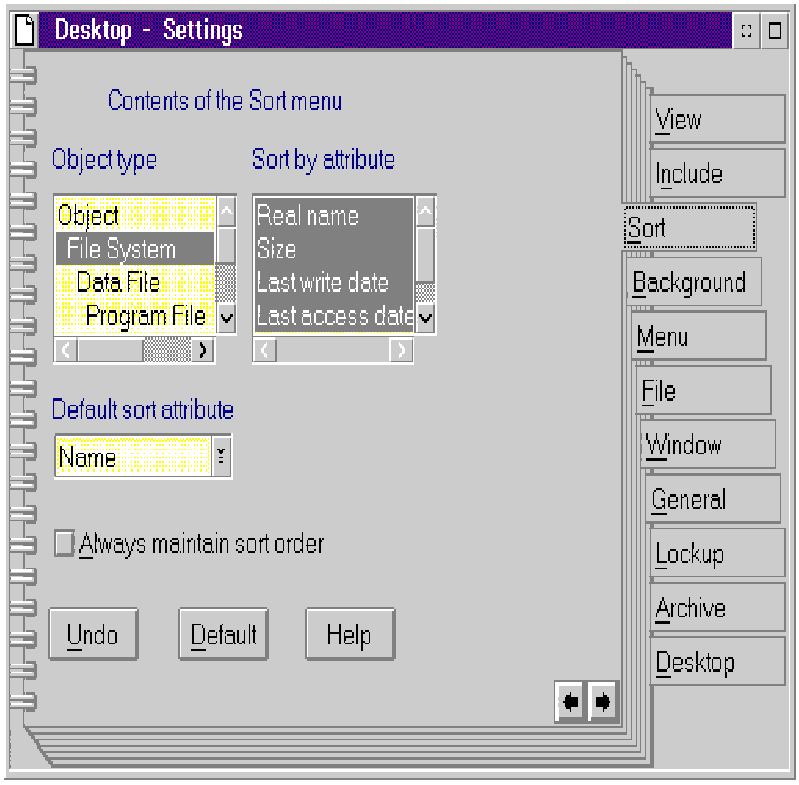
Specifies more criterias for the Find process. This brings up the same dialog window as the one used in "Changing Criteria".

Find

Starts the search process. After the Find process is started, a Search in progress dialog is displayed. When the search is completed a Find results window comes up that shows the name of all objects found.

Sorting Objects

You specify objects to sort on by selecting the Sort tab of the Settings notebook. The following figure shows the Sort dialog:



There are four elements to consider for sorting objects:

Element	Description
Object type	Defines the type of object to sort on.
Sort by attribute	Defines the attributes to sort on. The attributes are associated with the object type selected.
Default sort attribute	Specifies the system default sort attribute.
Always maintain sort order	Refreshes any Open views every time a folder's object is moved, copied, or shadowed inside or outside its folder.

To select the previous sort criteria, press the Undo push button. To select the default sort criteria, press the Default push button.

Two object Details fields introduced by WPObject are sortable criteria: Name and Type. WPFileSystem introduces eight other sortable criteria. The following table shows the name of each detail as shown on the object's context menu:

Object Details Field	Attribute
Name	Name
Type	Type
Real name	Real name
Size of the file	Size
Last modification date	Last write date
Last modification time	Last write time
Last access date	Last access date
Last access time	Last access time
Date of creation	Creation date
Time of creation	Creation time

WPObject and WPFileSystem are the only system classes that contribute Details fields to the Sort dialog of the Settings notebook. User-defined class may introduce their own Details fields as sort criteria.

The sort class for a particular container determines what class' sort Details fields will appear as active sort criteria on the container's Sort dialog of the Settings page. The default sort class for all containers is WPFileSystem. This sort class can be manually changed via the container's Settings page, and it can also be programmatically changed by using wpSetFldrSortClass.

The wpSetFldrSortClass and wpQueryFldrSortClass methods are needed to set both the Always maintain sort order checkbox on a container's Sort dialog, as well as the chosen sort field.

Note: The sort methods are explained in Folder Related Methods.

Object Errors: Error Handling Methods

The Workplace Shell provides methods which you can override to design your own error handling system, using error objects. This chapter provides information on error handling methods for Workplace Shell applications.

About Error Handling Methods

The WPObject class provides methods which can be overridden to design your own error handling system.

The wpQueryError method obtains the error identity that was set on an object by the last call to wpSetError. The wpQueryError method can be called at any time.

The wpSetError method sets the error identity on an object. Typically, this method is called prior to returning from a method that is unsuccessful. The wpQueryError method is then called to identify the error.

Similar to object errors, class errors can also be queried and set using wpclsQueryError and wpclsSetError.

The wpclsQueryError method obtains the error identity that was set on a class object by the last call to wpclsSetError. The wpclsQueryError method can be called at any time.

The wpclsSetError method sets the error identity on a class object. Typically, this method is called prior to returning from a class method that is unsuccessful. The wpclsQueryError method is then called to identify the error.

Using Error Handling Methods

This section describes, via sample code fragments, how you can use the error handling methods. Specifically the first sample code shows how wpQueryError retrieves the error identity that was set on an object. The second code fragment shows how to design your own error handling system by overriding wpSetError.

The following sample code shows how wpQueryError retrieves the error identity that was set on an object:

```
// METHOD: my_wpDrop
//
// DESCRIPTION:
//   SOMObject
//         WPObject
//               WPFileSystem
//                     WPFolder
//                           MyFolder
//
//   When an object is dropped on a folder,
//   this method is called on the target folder.
//
//   This sample code shows how wpQueryError retrieves
//   the error identity that was set on an object.
//
SOM_Scope MRESULT SOMLINK my_wpDrop(MyFolder *self,
```

```
                                    HWND      hwndCnr,
                                    PDRAGINFO pDragInfo,
                                    PDRAGITEM pDragItem)
{
  MRESULT       mr;        // Return value
  ULONG         ulError;   // Error code set on folder

  // Clear existing error code
  _wpSetError(self, NO_ERROR);
  // Call the parent to handle the drop
  mr = parent_wpDrop(self, hwndCnr, pDragInfo, pDragItem);

  // If parent returns an error, check to see if the disk is full
  if (mr == (MRESULT)RC_DROP_ERROR)
  {
    // Query the error code set on the folder
    ulError = _wpQueryError(self);
    // If the disk is full, display a message to the user
    if (ulError == ERROR_DISK_FULL)
    {
      WinMessageBox(HWND_DESKTOP,
                    NULLHANDLE,             // Owner Window
                    "Error Disk Full",      // Body of the message
                    "Unsuccessful Drop",    // Title of the message
                    0,                      // Message box id
                    MB_ERROR | MB_OK);      // Icon and button flags
    }
  }
  return mr;
}
```

The following sample code shows shows how to design your own error handling system by overriding wpSetError:

```
// METHOD: my_wpSetError
//
// DESCRIPTION:
//   Sets the error on an object.
//   User-defined errors can be handled here.
//
//   This sample code shows how to design your own
//   error handling system by overriding wpSetError.

SOM_Scope BOOL SOMLINK my_wpSetError(MyFolder *self,
                                     ULONG ulErrorId)
{
  // Check to see if this is a user-defined error
  if (ulErrorId > WPERR_USER)
  {
    // Handle user-defined errors here
  }
  // Call the parent to set the error
  return (parent_wpSetError(self, ulErrorId));
}
```

Object Information: Set/Query Methods

The Workplace Shell provides several set and query methods that you can use to set or to get object information, such as the default view of the object's in-use list, the object's icon, and the object's title. This chapter provides information for using object information methods. It describes how to define the behavior of Workplace Shell objects, using applicable styles.

About Set/Query Methods

The set and query methods are used to set and interrogate the properties of objects. The information associated with an object can be:

- Default help
- Default view
- Error set
- Handle
- Icon
- Icon data
- Identifier
- Settings page size
- Title.

As some methods relate to one another, they have been grouped into the following sections:

- General (including Class style and Object style)
- Data file
- Desktop
- Disk (File-System Device)
- File System
- Folder
- Icon
- Network
- Palette
- Power
- Printer
- Program reference and program file
- Shadow
- View.

Each section is divided into tables that are grouped as follows:

- Methods that obtain information and that are overridden
- Methods that obtain information but are not normally overridden
- Methods that set information and that are overridden
- Methods that set information but are not normally overridden.

Note: The help and error methods are explained in the following chapters:

Error handling methods	Object Errors: Error Handling Methods.
Help methods	Object Aid: Help Methods.

General Characteristics Methods

A program or Workplace Shell object would call the following methods to get general information about an object. A Workplace Shell object would override these methods to define its characteristics.

Method Name	Description
wpIdentify	Returns a string that is unique within a given container that identifies the specified object.
wpQueryConfirmations	Returns the set of confirmations that are set for this object.
wpQueryDefaultHelp	Returns the object's default help panel.
wpQueryNameClashOptions	Returns the set of options that are not available on the name clash dialog for this object.
wpQueryTitle	Returns the object's title.
wpQueryTrueStyle	Queries this object's current class true style.

wpclsQueryDefaultHelp	Returns the class default help panel. This help panel is used for all objects of this class. The default help panel can be changed for an individual object if: o The object overrides wpQueryDefaultHelp. o The wpSetDefaultHelp method is called. o The wpSetup method is called with the HELPPANEL keyword.
wpclsQuerySetting	Returns a settings value for an abstract class object. An abstract class object would override this method if it saved its class settings.
wpclsQueryStyle	Returns the class style. See Class Styles.
wpclsQueryTitle	Returns the class title. This title is used for all objects of this class, including the class template. The title can be changed for an individual object if: o The object overrides wpQueryTitle. o The wpSetTitle method is called. o The wpSetup method is called with the TITLE keyword, when the user edits the object's title on the General page of the Settings notebook, or when the user edits the object's title under the icon.

A program or Workplace Shell object would call the following methods to get general information about an object. These methods are not normally overridden.

Method Name	Description
wpFindTaskRec	An object is sent this message to return information specific to the task being performed in a task thread.
wpGetTrueClassName	Returns the class name for the given object.
wpQueryContainerFlagPtr	Returns the pointer to a flag indicating whether or not the object is in a container.
wpQueryCoreRecord	Returns the pointer to the object's MINIRECORDCORE data structure.
wpQueryDefaultIconPos	Returns the default icon position used when this object is first inserted into a folder.
wpQueryError	Returns the last error code set by calling the object's wpSetError. This is normally an error code being returned by a failing instance method.
wpQueryFolder	Returns the object pointer for the folder containing the

	object.
wpQueryHandle	Returns the object's unique persistent handle.
wpQueryObjectID	Returns the object's identifier. The object identifier is set by calling wpSetObjectID for the object or by calling wpSetup for the object with the OBJECTID keyword.
wpQueryScreenGroupID	Returns the screen group identifier of the application running for this object.
wpQueryStyle	Returns the object's current style flags. See Object Styles .
wpQueryStyle	Returns the object's style. See Object Styles.
wpclsQueryError	Returns the last error code set by calling the class' wpclsSetError. This is normally an error code that is returned by a failing class method.
wpclsQueryFolder	Returns the pointer to the folder object that corresponds to the specified file system path.
wpclsQueryObject	Returns the pointer to the object that corresponds to the specified persistent object handle.
wpclsQueryStyle	Returns the object class' style. See Class Styles.

A program or Workplace Shell object would call the following methods to set general information about an object. These methods are not normally overridden.

Method Name	Description
wpModifyStyle	Sets or clears individual flags within the object's style. This method call is an atomic operation as only the specified style flags are changed, even if other running threads try to modify the style flags at the same time. See Object Styles.
wpSetDefaultHelp	Sets the object's default help panel.
wpSetDefaultIconPos	Sets the object's default icon position which is used when it is inserted into a folder.
wpSetError	Sets the current error code for the object. This method is normally called by an instance method to pass an error code back to the caller.
wpSetObjectID	Sets the object's identifier.
wpSetStyle	Sets the object's style. This method exists only for compatibility with old objects. All new or changed objects should use

	wpModifyStyle instead. See Object Styles.
wpSetTaskRec	Adds, replaces, or deletes a task data block.
wpSetTitle	Sets the object's title.
wpclsSetError	Sets the current error code for the class. This method is normally called by a class method to pass an error code back to the caller.

A program or Workplace Shell object would call the following method to set general information about an object. A Workplace Shell object would override this method to define its characteristics.

Method Name	Description
wpclsQueryInstanceClass	Returns the name of a class that this object can become.
wpclsSetSetting	Sets a settings value for an abstract class object. An abstract class object would override this method if it wants to save its class settings.

Class Styles

Workplace Shell object classes have styles that define the behavior of all objects of that class. The object class style is defined by wpclsQueryStyle. When an object is created, the object's style is initially inherited from the class. The following table lists the class style flags:

Style Name	Description
CLSSTYLE_NEVERCOPY	Objects of this class cannot be copied.
CLSSTYLE_NEVERDELETE	Objects of this class cannot be deleted.
CLSSTYLE_NEVERDRAG	Objects of this class cannot be dragged.
CLSSTYLE_NEVERLINK	Objects of this class cannot have shadows.
CLSSTYLE_NEVERMOVE	Objects of this class cannot be moved.
CLSSTYLE_NEVERPRINT	Objects of this class cannot be printed.
CLSSTYLE_NEVERRENAME	Objects of this class cannot be renamed.
CLSSTYLE_NEVERSETTINGS	Objects of this class cannot be opened in Settings view.
CLSSTYLE_NEVERTEMPLATE	This class does not have a template. Also, objects of this class will not have a Create another option in their context menu.
CLSSTYLE_NEVERVISIBLE	Objects of this class are never visible.

Object Styles

Workplace Shell objects have styles that define their behavior. The object class style is defined by wpQueryStyle. When an object is created, the object's style is initially inherited from the class. The wpQueryStyle method can modify the style flags to tailor the object. Also, wpModifyStyle can be used to set or reset individual style flags for the object. The following table lists the object style flags:

Style Name	Description
OBJSTYLE_NOCOPY	The object cannot be copied. The Copy option is not valid for a direct manipulation operation (Pickup and Drop) and there is no Copy option in the object's context menu. This option is inherited from the class' CLSSTYLE_NEVERCOPY style.
OBJSTYLE_NODELETE	The object cannot be deleted. The object cannot be dragged to the shredder and there is no Delete option in the object's context menu. This option is inherited from the class' CLSSTYLE_NEVERDELETE style.
OBJSTYLE_NODRAG	The object cannot be copied or moved. Direct manipulation operations (Pickup and Drop) are not valid for this object and there are no Copy, Move, or Pickup options in the object's context menu. This option is inherited from the class' CLSSTYLE_NEVERDRAG style.
OBJSTYLE_NODROPON	No other object can be dropped on this object using a direct manipulation operation (Pickup and Drop).
OBJSTYLE_NOLINK	The object cannot have shadows. The Link option is not valid for a direct manipulation operation (Pickup and Drop) and there is no Create shadow option in the object's context menu. This option is inherited from the class' CLSSTYLE_NEVERLINK style.
OBJSTYLE_NOMOVE	The object cannot be moved. The Move option is not valid for a direct manipulation operation (Pickup and Drop) and there is no Move option in the object's context menu. This option is inherited from the class' CLSSTYLE_NEVERMOVE style.
OBJSTYLE_NOPRINT	The object cannot be printed. The object cannot be dragged to a Printer object and there is no Print option in the object's context menu. This option is inherited from the class' CLSSTYLE_NEVERPRINT

	style.
OBJSTYLE_NORENAME	This object cannot be renamed. Its title cannot be changed either by directly editing the title under the icon or by changing the title on the General page of the Settings notebook.
OBJSTYLE_NOSETTINGS	This object cannot be opened in Settings view. There is no Settings option in the object's context menu. This option is inherited from the class' CLSSTYLE_NEVERSETTINGS style.
OBJSTYLE_NOTDEFAULTICON	The icon associated with this object is not the class default icon. Note: You should never modify this style option because the Workplace Shell uses it for the following reasons: o When you copy an object, the Workplace Shell determines when to create a copy of the object's icon. o When the Workplace Shell frees an icon. If this style flag is set incorrectly, the object's icon could disappear or could not be freed when necessary, thereby wasting system memory.
OBJSTYLE_TEMPLATE	This object is a template.

Data File Related Methods

A program or Workplace Shell object would call the following methods to get information about a data file object. These methods are not normally overridden.

Method Name	Description
wpQueryAssociatedFileIcon	Returns the handle of the icon for the WPProgram or WPProgramFile object associated with the data file object.
wpQueryAssociatedProgram	Returns the pointer to the WPProgram or WPProgramFile object associated with the data file object.
wpQueryPointerFromContents	Returns a pointer to shared memory containing the contents of the WPDataFile object.
wpclsEnumClipboardClasses	Enumerates through all the object classes that a particular rendering mechanism supports.

A program or Workplace Shell object would call the following method to query information about a data file object. This method may be overridden to force the association of a specific icon with a data file.

Method Name	Description
wpQueryHandleFromContents	Returns a handle to a known system data type based on a file's contents. This is an abstract method; subclasses that describe files for which a system handle type exist (such as WPIcon or WPBitmap) should subclass this method.

A program or Workplace Shell object would call the following method to set information about a data file object. This method may be overridden to force the association of a specific icon with a data file.

Method Name	Description
wpSetAssociatedFileIcon	Change the icon associated with the data file object.
wpSetContentsFromHandle	Sets a file's contents based on a handle to a known system data type. This is an abstract method; subclasses that describe files for which a system handle type exist (such as WPIcon or WPBitmap) should subclass this method.
wpSetContentsFromPointer	Sets a WPDataFile object's contents based on a pointer to shared memory.

A program or Workplace Shell object would call the following method to render information about a data file object.

Method Name	Description
wpRenderFromClipboard	Opens the clipboard and retrieves the type of data specified in the clipboard format passed in, then writes the clipboard data to a file.

A program or Workplace Shell object would call the following method to add a clipboard association to a data file object.

Method Name	Description
wpclsAddClipboardAssoc	Adds an association between a clipboard format and a particular class name.

Desktop Related Methods

A program or Workplace Shell object would call the following methods to get information about a Desktop object. These methods are not normally overridden.

Method Name	Description
wpIsCurrentDesktop	Returns an indication of whether or not the Desktop object is the active Desktop.
wpclsQueryActiveDesktop	Returns the pointer to the active Desktop object.
wpclsQueryActiveDesktopHWND	Returns the window handle for the active Desktop.
wpQueryAutoLockup	Returns the current value of the automatic lockup option.
wpQueryLockupAutoDim	Returns the current value of the lockup auto-dim option.
wpQueryLockupBackground	Returns the current values for the lockup background.
wpQueryLockupFullScreen	Returns the current value of the lockup full-screen option.
wpQueryLockupOnStartup	Returns the current value of the lockup on the startup option.
wpQueryLockupTimeout	Returns the current timeout value for automatic lockup.
wpclsQueryObjectFromFrame	Returns the pointer to the object associated with the specified frame window.

A program or Workplace Shell object would call the following methods to get information about a Desktop object.

Method Name	Description
wpQueryDefaultDropOp	Queries the default drop operation of objects that are dropped on the Desktop. The default is set on the Desktop page of the Desktop's Settings notebook. This method can be overridden to control the Desktop's default drop operation

A program or Workplace Shell object would call the following methods to get information about a Desktop object. These methods are not normally overridden.

Method Name	Description
wpSetAutoLockup	Sets the value of the automatic lockup option.
wpSetDefaultDropOp	Sets the default drop operation of objects that are dropped on the Desktop. The default is set on the Desktop page of the Desktop's Settings notebook.
wpSetLockupAutoDim	Sets the value of the lockup auto-dim option.
wpSetLockupBackground	Sets the current values for the lockup background.
wpSetLockupFullScreen	Sets the value of the lockup full-screen option.
wpSetLockupOnStartup	Sets the value of the lockup on the startup option.

wpSetLockupTimeout	Sets the timeout value for automatic lockup.

Disk (File-System Device) Related Methods

A program or Workplace Shell object would call the following methods to set information about a disk object. These methods are not normally overridden.

Method Name	Description
wpQueryDriveLockStatus	Returns the lock status of a disk object.
wpQueryLogicalDrive	Returns the logical drive number for a disk object.
wpQueryRootFolder	Returns the root folder object for the logical drive represented by a disk object.

A program or Workplace Shell object would call the following method to set information about a disk object. A disk object would override this method to define its characteristics.

Method Name	Description
wpSetCorrectDiskIcon	Sets the correct icon for the disk object.

File System Related Methods

A program or Workplace Shell object would call the following methods to get information about a file system object. These methods are not normally overridden.

Method Name	Description
wpQueryAttr	Returns the file attributes of the file system object.
wpQueryCreation	Returns the creation date and time of the file system object.
wpQueryDateInfo	Returns the file system object's: o Creation date and time o Last access date and timeo o Last update date and time.
wpQueryDisk	Returns the pointer to the disk object for the drive on which the file system object resides.
wpQueryEASize	Returns the size, in bytes, of the extended attributes of the file system object.

wpQueryEASupport	Determines whether extended attributes are supported for the specified file system object.
wpQueryFilename	Returns the name of the file system object.
wpQueryFileSize	Returns the size, in bytes, of the file system object.
wpQueryLastAccess	Returns the last access date and time of the file system object.
wpQueryLastWrite	Returns the last write date and time of the file system object.
wpQueryRealName	Returns the real name of the file system object. For file systems that do not support long file names, such as the FAT file system which allows 8 characters for the file name and 3 for the extension, the real name may be different than the object's title.
wpQueryRefreshFlags	Queries the refresh flags for a file system object. The refresh flags consist of a DIRTYBIT and a FOUNDBIT that are used to allow refresh to detect deleted files and new files after asking a folder to re-populate.
wpQueryType	Returns the type of the file system object.
wpclsQueryAwakeObject	Returns the pointer to the file system object corresponding to the specified path name, if the file system object is awake.
wpclsQueryInstanceFilter	Returns a string containing the filter for files that comprise the object class.
wpclsQueryInstanceType	Returns a string containing the .TYPE attribute for the files that comprise the object class.
wpclsQueryObjectFromPath	Returns the pointer to an object which represents the given file or directory.

A program or Workplace Shell object would call the following methods to set information about a file system object. These methods are not normally overridden.

Method Name	Description
wpSetAttr	Changes the file attributes of the file system object.
wpSetDateInfo	Changes the file system object's: o Creation date and time o Last access date and timeo o Last update date and time. Note: This method changes only the dates and times kept in the persistent data of the file

	system object. It does not change the actual dates and times kept by the underlying file system.
wpSetFileSizeInfo	Changes the file size and extended attribute size of the file system object. Note: This method changes only the sizes kept in the persistent data of the file system object. It does not change the actual sizes used by the underlying file system.
wpSetRealName	Changes the real name of the file system object. This is the name of the object maintained by the underlying file system.
wpSetTitleAndRenameFile	Changes the file system object's title and real name, so that they match.
wpSetType	Changes the .TYPE attribute of the file system object.

Folder Related Methods

A program or Workplace Shell object would call the following methods to get information about a folder object. These methods are not normally overridden.

Method Name	Description
wpInitIconPosData	Reads the .ICONPOS extended attribute (EA) for the folder directory to determine where to position the object icons within the folder.
wpIsDetailsColumnVisible	Returns the visibility state of a specified Details view column of the folder object.
wpIsSortAttribAvailable	Returns an indication of whether or not a sort attribute is available for a column of the Details view of the folder object.
wpQueryContent	Returns the pointer to an object contained in the folder. This method may be called within a loop to enumerate all of the objects contained in the folder. Note: The folder must be awake for this method to return the pointers to all objects contained in the folder.
wpQueryFldrAttr	Returns the folder object's attributes.
wpQueryFldrBackground	Returns the current values for the folder background.

wpQueryFldrDetailsClass	Returns the class for which data is displayed in the Details view of the folder object.
wpQueryFldrFlags	Returns the current state of the folder object's flags.
wpQueryFldrFont	Returns the name of the font used for the specified view of the folder object.
wpQueryFldrSort	Returns the sort attribute used for the specified view of the folder object.
wpQueryFldrSortClass	Returns the object class for which sorting is performed for the folder object.
wpQueryIconPosition	Finds the saved icon postion for this record.
wpQueryIconTextBackgroundColor	Returns the current color being used for the icon text background.
wpQueryIconTextColor	Returns the current color being used for the icon text in the specified view.
wpQueryIconTextVisibility	Returns the current state of icon text visibility for the specified view.
wpQueryIconViewPos	Returns a pointer to the IconViewPos array of the folder object.
wpQueryMenuBarVisibility	Returns the visibility state of the folder's menu bar.
wpQueryMenuStyle	Returns the current menu style, which is either long or short.
wpQueryNextIconPos	Returns the next position at which an icon is inserted into the folder object.
wpQueryOpenFolders	Allows for the enumeration of all open folders in the system.
wpQueryShadowTextColor	Returns the current color being used for the shadow text in the specified view.
wpclsQueryOpenFolders	Returns the pointer to an open folder object. This method may be called in a loop to enumerate all currently open folders.

A program or Workplace Shell object would call the following methods to set information about a folder object. These methods are not normally overridden.

Method Name	Description
wpModifyFldrFlags	Sets or clears individual flags within the folder object. Note: This method is an atomic operation as only the specified flags are changed, even if other running threads

	try to modify the flags at the same time.
wpSetDetailsColumnVisibility	Changes the visibility state of a specified column of data for the Details view of the folder object.
wpSetFldrAttr	Changes the folder object's attributes.
wpSetFldrBackground	Sets the values for the folder background.
wpSetFldrDetailsClass	Changes the object class for which data is displayed in a Details view of the folder object.
wpSetFldrFlags	Changes the folder object's flags. Note: This method exists only for compatibility with old objects. All new or changed objects should use wpModifyFldrFlags instead.
wpSetFldrFont	Changes the font used for the specified view of the folder object.
wpSetFldrSort	Changes the sort attribute for the specified view of the folder object.
wpSetFldrSortClass	Changes the object class for which data is sorted for the folder object.
wpSetFolder	Sets this object to reside in the specified folder.
wpSetIconTextBackgroundColor	Sets the color to be used for the icon text background.
wpSetIconTextColor	Sets the color to be used for the icon text for the specified view.
wpSetIconTextVisibility	Sets the icon text visibility for the specified view.
wpSetMenuBarVisibility	Sets the visibility state of the folder's menu bar.
wpSetNextIconPos	Changes the next position at which an icon is inserted into the folder object. This method can be used to ensure that objects inserted into the folder are displayed where you want them.
wpSetRefreshFlags	Sets the refresh flags for a file system object. The refresh flags consist of a DIRTYBIT and a FOUNDBIT that are used to allow refresh to detect deleted files and new files, after asking a folder to re-populate.
wpSetShadowTextColor	Sets the color to be used for the shadow text for the specified view.
wpSetSortAttribAvailable	Changes the indication of whether or not a sort attribute is available for a column of a Details view of the folder object.

A program or Workplace Shell object would call the following methods to flush notification messages or free icon position data for a folder object. These methods are not normally overridden.

Method Name	Description
wpFlushNotifications	Forces all pending file system notifications for a folder to be processed immediately.
wpFreeIconPosData	Frees the .ICONPOS extended attribute (EA) information that is allocated by wpInitIconPosData.

Icon Related Methods

A program or Workplace Shell object would call the following methods to get information about an object's icons. A Workplace Shell object would override these methods to define its icons.

Method Name	Description
wpQueryHandleFromContents	Returns a handle to a known system data type based on a file's contents. This is an abstract method; subclasses that describe files for which a system handle type exist (such as WPIcon or WPBitmap) should subclass this method.
wpQueryIconData	Returns the information that defines the object's icon. This overrides the class default icon defined by wpclsQueryIconData.
wpclsQueryIconData	Returns the information that defines the class default icon. This icon is associated with every object of this class. This icon can be changed for an individual object if: o The object overrides wpQueryIconData. o The wpSetIconData method is called. o The wpSetup method is called with the ICONFILE or ICONRESOURCE keyword, or if the user edits the icon on the General page of the Settings notebook. For a folder class, this is the closed folder icon.
wpclsQueryIconDataN	Returns the information that defines the nth class default animation icon. Note: This method is available only for a folder class. It defines the default open folder icon for all objects of this class. This icon can

	be changed for an individual object if wpSetup is called with the ICONNFILE or ICONNRESOURCE keyword, or if the user edits the animation icon on General page 2 of the Settings notebook.

A program or Workplace Shell object would call the following methods to get information about an object's icons. These methods are not normally overridden by the object.

Method Name	Description
wpQueryIcon	Returns the handle of the object's icon.
wpclsQueryIcon	Returns the handle of the class default icon. For a folder class, this method returns the handle of the class' default closed folder icon.
wpclsQueryIconN	Returns the handle of the nth class default animation icon. This method is available only for a folder class. It returns the handle of the class' default open folder icon.

A program or Workplace Shell object would call the following methods to change the icon for an object. These methods are not normally overridden by the object.

Method Name	Description
wpSetIcon	Sets the handle of the object's icon. This handle is not saved in the persistent data of the object, so it has effect until the object goes dormant. This method affects all open views of the object. For a folder object, wpSetIcon affects the icon displayed: o In the title bar. o On the Toolbar. o On the General page 1 of the Settings notebook. o In all containers if the object is closed. If the object is open, all containers continue to show the object's animation icon.
wpSetIconData	Sets the definition for the object's icon. This icon definition is saved with the persistent data of the object. It permanently changes the object's icon. This method automatically calls wpSetIcon to update the visual display of the object's icon.
wpSetProgIcon	Sets the visible icon associated with the program reference or program file object. This method is available only for a program reference or program file object.
wpclsSetIcon	Sets the handle of the default class icon.

wpclsSetIconData	Sets the definition of the default class icon.

Image File Related Methods

A program or Workplace Shell object would call the following methods to get information about an image file object. These methods are not normally overridden.

Method Name	Description
wpQueryBitmapData	Returns a pointer to the bitmap data for the image file.
wpQueryBitmapHandle	Returns a handle to the bitmap that can be displayed showing the contents of the image file and a handle to the palette to be used when displaying the bitmap.
wpQueryBitmapInfoHeader	Returns a pointer to the bitmap information header for the image file.

A program or Workplace Shell object would call the following method to set information about an image file object.

Method Name	Description
wpSetBitmapData	Sets the bitmap data for the image file. This method will convert the bitmap data to the appropriate format and update the actual image data file.

A program or Workplace Shell object would call the following method to read or write information about an image file object.

Method Name	Description
wpReadImageFile	Reads the image data for the image file and stores it in the instance data for the object. If the image data has already been read, this method simply returns TRUE. It does not refresh the image data.
wpWriteImageFile	Writes the image data for the image file from the instance data.

Network Related Methods

A program or Workplace Shell object would call the following methods to get information about a network object. These methods are not normally overridden.

Method Name	Description
wpQueryNetIdentity	Returns the fully qualified network name for a network group object.
wpQueryObjectNetId	Returns the network ID of the shared directory object that this object is linked to.
wpQuerySrvrIdentity	Returns the fully qualified network name for this server object.

A program or Workplace Shell object would call the following methods to set information about a network object. These methods are not normally overridden.

Method Name	Description
wpSetNetIdentity	Sets the fully qualified network name for a network group object.
wpSetObjectNetId	Sets the network ID of the shared directory object that this object is linked to.

Palette Related Methods

A program or Workplace Shell object would call the following methods to get information about a palette object. A palette object would override these methods to define its characteristics.

Method Name	Description
wpQueryPaletteHelp	Returns the help panel ID that is displayed when the Help push button is pressed in an open palette view.
wpclsQueryEditString	Returns the title for the Edit push button.

A program or Workplace Shell object would call the following method to get information about a palette object. This method is normally not overridden.

Method Name	Description
wpQueryPaletteInfo	Returns information about the palette.

A program or Workplace Shell object would call the following method to set information about a palette object. This method is normally not overridden.

Method Name	Description
wpSetPaletteInfo	Changes the current information about the palette.

Power Related Methods

A program or Workplace Shell object would call the following methods to get information about a power object. A power object would override these methods to define its characteristics.

Method Name	Description
wpQueryAutoRefresh	Returns the current state (enabled or disabled) for automatic refresh of the power object's status window.
wpQueryRefreshRate	Returns the current refresh rate (in minutes) for the power object's status window when automatic refresh is enabled.

A program or Workplace Shell object would call the following methods to get information about a power object. These methods are normally not overridden.

Method Name	Description
wpQueryDefStatusView	Returns the default view for the power object when "Advanced power management" (APM) is enabled.
wpQueryPowerConfirmation	Returns the current state (enabled or disabled) for the power object's confirmation message. This message is optionally displayed when power state changes are requested from its context menu.
wpQueryPowerManagement	Returns the current state (enabled or disabled) for "Advanced power management" (APM).

A program or Workplace Shell object would call the following methods to set information about a power object. These methods are normally not overridden.

Method Name	Description
wpSetAutoRefresh	Enables or disables automatic refresh of the power object's status window.
wpSetDefStatusView	Sets the default status view for the power object when "Advanced power management" is enabled.
wpSetPowerConfirmation	Enables or disables the display of the power object's confirmation message when power state changes are requested from its context menu.
wpSetPowerManagement	Enables or disables "Advanced power management".

wpSetRefreshRate	Sets the refresh rate for the power object's status window when automatic refresh is enabled.

Printer and Print Job Related Methods

A program or Workplace Shell object would call the following methods to get information about a printer or print job object. These methods are normally not overridden.

Method Name	Description
wpQueryComputerName	Returns the name of the computer on which the printer object exists.
wpQueryJobFile	Returns the name of the spool file for the print job object.
wpQueryJobId	Returns the job ID for the print job object.
wpQueryJobType	Returns the data type for the print job object.
wpQueryLocalAlias	Returns the name of the local print queue for a remote printer object.
wpQueryNetworkId	Returns the fully qualified network ID of the remote printer queue for a remote printer object.
wpQueryPrinterName	Returns the name of the print queue for this printer object. For a remote printer object, this is the computer's printer queue name on which the printer object exists. The name of the local print queue can be obtained by calling wpQueryLocalAlias.
wpQueryPrintObject	Returns a pointer to the printer object to which the print job is associated with.
wpQueryQueueOptions	Returns the printer object's queue options.
wpQueryRemoteOptions	Returns the printer object's remote options.

A program or Workplace Shell object would call the following methods to set information about a printer or print job object. These methods are normally not overridden.

Method Name	Description
wpSetComputerName	Sets the name of the computer on which the printer object resides.
wpSetDefaultPrinter	Sets the printer object as the default printer.
wpSetJobProperties	Sets the print job properties.
wpSetPrinterName	Sets the name of the printer

	object.
wpSetQueueOptions	Sets the printer object's queue options.
wpSetRemoteOptions	Sets the printer object's remote options.

Program Reference and Program File Related Methods

A program or Workplace Shell object would call the following methods to get information about a program reference or a program file object. A program reference or program file object would override these methods to define its characteristics.

Method Name	Description
wpQueryAssociationFilter	Returns the list of filters that the program reference or program file object is associated to.
wpQueryAssociationType	Returns the list of types that the program reference or program file object is associated to.
wpQueryProgramAssociations	Returns the list of filters or types for the program reference or program file object.

A program or Workplace Shell object would call the following method to get information about a program reference or a program file object. This method is not normally overriden.

Method Name	Description
wpQueryProgDetails	Returns the program details for the program reference or program file object.
wpQueryWarnBeforeStart	Queries the program reference's Warn Before Start flag. If this flag is set, put up a warning dialog before starting the program.
wpQueryWarnBeforeStart	Queries the program reference's Warn Before Start flag. If this flag is set, put up a warning dialog before starting the program.

A program or Workplace Shell object would call the following methods to set information about a program reference or a program file object. These methods are not normally overridden.

Method Name	Description
wpSetAssociationFilter	Changes the list of filters that the program reference or program file object is associated to.
wpSetAssociationType	Changes the list of types that the program reference or program file object is associated to.

wpSetProgDetails	Changes the program details for the program reference or program file object.
wpSetProgramAssociations	Changes the list of filters or types that the program reference or program file object is associated to.
wpSetWarnBeforeStart	Sets the program reference's Warn Before Start flag. If this flag is set, put up a warning dialog before starting the program.
wpSetWarnBeforeStart	Sets the program reference's Warn Before Start flag. If this flag is set, put up a warning dialog before starting the program.

Shadow Related Methods

A program or Workplace Shell object would call the following method to get information about a shadow object. This method is normally not overridden.

Method Name	Description
wpQueryShadowedObject	Returns a pointer to the object this shadow is linked to.

A program or Workplace Shell object would call the following methods to set information about a shadow object. These methods are normally not overridden.

Method Name	Description
wpSetLinkToObject	Changes a shadow object to link to a new object.
wpSetShadowTitle	Changes the title of a shadow object without changing the title of the object the shadow is linked to.

View Related Methods

A program or Workplace Shell object would call the following methods to get information about a view of an object. A Workplace Shell object would override these methods to define the characteristics of its open views.

Method Name	Description
wpQueryButtonAppearance	Returns which button (minimize, hide, or the system default) appears in the frame control for an open view of the object.

wpQueryConcurrentView	Returns an indication of whether or not this object allows multiple concurrent views to be opened.
wpQueryDefaultView	Returns the view of the object that is opened when the default view is opened.
wpQueryMinWindow	Returns the behavior of the object when it is minimized (Minimize window to Desktop, Minimize window to viewer, or use the system default behavior).
wpclsQueryButtonAppearance	Returns the class default for which button (minimize, hide, or the system default) appears in the frame control for an open view of any object of this class.
wpclsQueryDefaultView	Returns the class default of which view of the object is opened when the default view is opened. The default view is associated with every object of the class. The default view can be changed for an individual object if: o The object overrides wpQueryDefaultView. o The wpSetDefaultView method is called. o The wpSetup method is called with the DEFAULTVIEW keyword.
wpclsQuerySettingsPageSize	Returns the size of the Settings notebook pages displayed for the object class.

A program or Workplace Shell object would call the following methods to set the characteristics for the open views of the object. These methods are not normally overridden.

Method Name	Description
wpSetButtonAppearance	Defines which button (minimize, hide, or the system default) appears in the frame control for an open view of the object. This value is normally returned by wpQueryButtonAppearance.
wpSetConcurrentView	Defines whether or not the object allows multiple concurrent views to be opened. This value is normally returned by wpQueryConcurrentView.
wpSetDefaultView	Defines which view of the object is opened when the default view is opened. This value is normally returned by wpQueryDefaultView.
wpSetMinWindow	Defines the behavior of the object when it is minimized (Minimize window to Desktop, Minimize window to viewer, or use the system default behavior). This value is normally returned by

```
                                wpQueryMinWindow.

 wpclsSetSettingsPageSize       Sets the size of the Settings
                                notebook pages displayed for
                                the object class.
```

Using Set/Query Methods

The set and query methods can be divided into four different groups of methods:

- Instance query methods
- Instance set methods
- Class query methods
- Class set methods.

This section describes each of them and provides sample code fragments. It also explains their role, and whether or not they are used to override methods.

Instance Query Methods

The instance query methods are used to define the properties of an instance of your object. You would normally override these methods to define the unique properties of the instance of the object. For example, you might override wpclsQueryStyle if you want to prevent all instances of your object from being deleted. However, you would override wpQueryStyle to prevent only certain instances of this object from being deleted, as shown in the following sample code:

```
ULONG SOMLINK myfdr_wpQueryStyle(MyFolder *somSelf)
{
  ULONG ulStyle;
  ULONG ulAttr;

  MyFolderMethodDebug("MyFolder","myfdr_wpQueryStyle");
  ulStyle = parent_wpQueryStyle(somSelf);
  ulAttr  = _wpQueryAttr( somSelf );

  if (ulAttr & FILE_READONLY)
  {
    ulStyle |= OBJSTYLE_NODELETE;
  }
  return ulStyle;
}
```

Instance Set Methods

The instance set methods are used to modify the properties of a specific object instance. You would not normally override these methods but would call these methods to change the behavior of the corresponding instance query method or the behavior of the object. For example, you might call wpSetIcon to change the current icon for your object, as shown in the following sample code:

```
    .
    .
    .

/* Load the icon from the resource file */
hptrIcon = WinLoadPointer(HWND_DESKTOP,vhmod,IDP_Folder3);

/* Make this the current icon for the object */
```

```
_wpSetIcon(somSelf,hptrIcon);

    .
    .
    .
```

Class Query Methods

The class query methods are used to define the properties of an object class. You would normally override these methods to define the unique properties of the class object. For example, you would override wpclsQueryTitle to provide a unique title for your object class. If you were defining a class called MyFolder, as a subclass of WPFolder, you might supply the following method to make the default title for all instances (including the template) of the MyFolder object "My Folder".

```
PSZ  SOMLINK myfdrM_wpclsQueryTitle(M_MyFolder *self)
{
  M_MyFolderMethodDebug("M_MyFolder","myfdrM_wpclsQueryTitle");
  return ("My Folder");
}
```

If you want to provide a unique icon for all instances of your class, you would override wpclsQueryIconData, as shown in the following sample code:

```
ULONG SOMLINK myfdrM_wpclsQueryIconData(M_MyFolder *self,
                                        PICONINFO pIconInfo)

{
  M_MyFolderMethodDebug("M_MyFolder","myfdrM_wpclsQueryIconData");
  if (pIconInfo)
  {
    /*
     *  vhmod is a global variable containing the
     *  module handle for your .DLL
     *  IDP_MyFolder is the resource number for the class icon
     */
    pIconInfo->fFormat = ICON_RESOURCE;
    pIconInfo->hmod    = vhmod;
    pIconInfo->resid   = IDP_MyFolder;
  }
  return (sizeof(ICONINFO));
}
```

If your class is a subclass of WPFolder (as in the above example) and you want to provide an animation icon, that is the icon that is displayed when an instance of your folder object is opened, you would override wpclsQueryIconDataN, as shown in the following sample code:

```
ULONG SOMLINK fdrM_wpclsQueryIconDataN(M_MyFolder *self,
                                       PICONINFO pIconInfo,
                                       ULONG ulIconIndex)

{
  ULONG ulReturn;

  M_MyFolderMethodDebug("M_MyFolder","myfdrM_wpclsQueryIconDataN");
  switch (ulIconIndex)
  {
    case 0:
    /* Return the information for the closed folder icon */
    ulReturn = _wpclsQueryIconData(self,pIconInfo);
    break;
    case 1:
    /* Return the information for the open folder icon */
    if (pIconInfo)
    {
      pIconInfo->fFormat = ICON_RESOURCE;
      pIconInfo->hmod    = vhmod;
      pIconInfo->resid   = IDP_MyFolder2;
```

```
    }
    ulReturn = sizeof(ICONINFO);
    break;

    default:
    /* Invalid index value specified */
    ulReturn = 0;
  }
  return (ulReturn);
}
```

Class Set Methods

The class set methods are used to modify the properties of an object class. You would not normally override these methods but would call these methods to change the behavior of the corresponding class query method. For example, you might call wpclsSetError to define an error code to be returned by the next call to wpclsQueryError, as shown in the following sample code:

```
/* Set the class error code */
_wpclsSetError(_MyFolder,0x1234);
    .
    .
    .
/* Retrieve the last class error code */
ulLastError = _wpclsQueryError(_MyFolder);
```

Object Initialization and Termination: Setup/Cleanup Methods

The Workplace Shell provides methods that you can use to setup the characteristics and behaviors of an object. Likewise, it also provides methods that you can use to cleanup after an object is no longer in use. This chapter provides information on setup and cleanup methods that are available in the Workplace Shell.

About Setup/Cleanup Methods

Setup and cleanup methods support object initialization and termination. The setup methods are used to set up object characteristics during creation of the object. The cleanup methods are used to ensure that memory and other resources allocated to an object are freed when the object is no longer in use.

Create the Object

When an object is created, by using WinCreateObject for example, wpclsNew is called. You will likely not be concerned with this method because it performs functions such as allocating memory for the object, getting SOM to initialize the object, and creating the persistent image of the object as file system attributes or .INI file entries. You will be interested in methods that it invokes. These include wpInitData and wpSetupOnce.

Initialize the Object's Instance Data

The wpInitData method is called from WPObject's override of somDefaultInit immediately after the memory for the object has been allocated. The allocated memory is zero filled. If the zero initialization is not adequate for the object instance data for your class, override this method and set the instance variable values appropriately. The wpInitData method is called whenever the object is instantiated, whether it be during initial object creation or awakening it from dormancy. In the latter case, wpRestoreState is invoked after wpInitData.

Setup the Object's Characteristics and Behaviors

The wpSetupOnce method is called during initial object creation following wpInitData and the creation of the persistent image. It is not called when an object is awakened from dormancy. The primary purpose of wpSetupOnce is to:

- Call wpSetup to set the object characteristics and behaviors based on the setup string input to WinCreateObject.
- Perform non-setup string initialization that cannot be done during the wpInitData processing.

The non-setup string initialization may have been deferred because it required existence of the persistent image or because it was data other than instance data that required initialization. Override wpSetupOnce only if your class has non-setup string initialization to perform because WPObject wpSetupOnce calls wpSetup. Override wpSetup if your object class has defined unique setup string keywords or keyword values.

The wpSetup method is also invoked as a result of WinSetObjectData being called on an existing object. The wpScanSetupString method exists as a helper routine for parsing values from the setup string.

Setup and Query the Object's State

One of the last steps of object creation or awakening is setting the object to the initialized state. The wpIsObjectInitialized method queries the object for this indication.

The last step is the invocation of wpObjectReady. This is a notification to the object indicating completion of the creation or awakening operation. Override this method if you want to receive this notification. The wpObjectReady method indicates when it is safe to call wpSaveDeferred on a newly instantiated object.

Uninitialize the Object

The wpUnInitData method is the opposite of wpInitData. It is called by WPObject's override of somDestruct just prior to freeing the object's memory, during the process of making the object dormant or destroying it. The object's instance data, and memory allocated via wpAllocMem, are automatically deallocated. Override this method if your object has allocated additional resources since the last time it was awakened.

Destroy the Object

The wpFree method is the opposite of wpclsNew. It destroys the persistent image of the object and frees the object's memory. Override this method to free resources you have allocated that are associated with the persistent image of the object.

Workplace Shell Setup Strings

Classes can define keynames and values that affect the behavior of their objects. Use keynames with setup methods as follows:

- To set a certain behavior during object creation, specify the keyname in the setup string of WinCreateObject.
- To change the behavior of an existing object, specify the keyname in the setup string of WinSetObjectData.

Because keynames have default values, setup strings are not required for these calls.

Every class may define its own set of keynames and values. The *Workplace Shell Reference* contains the definition of keynames for all classes. Keynames and the values supported by the WPObject class are listed in the following table:

Keyname	Value	Description
CCVIEW	DEFAULT	Uses the default value of the concurrent view setting of the system, when the user selects open. This is the default value.
	YES	Creates new views of the object every time the user selects open.
	NO	Resurfaces open views of object, when the user selects open.
DEFAULTVIEW	DEFAULT	Sets the default open view to the object's class default view as returned by wpclsQueryDefaultView. This is the default value.
	SETTINGS	Sets the default open view to the Settings view.
	ID	Sets the default open view to the "id" of a user-added view (0 - 9).
HELPLIBRARY	filename	Sets the help library.
HELPPANEL	id	Sets the object's default help panel. This is equivalent to calling wpSetDefaultHelp.
HIDEBUTTON	YES	Views of this object have a hide button as opposed to a minimize button.
	NO	Views of this object have a minimize button as opposed to a hide button. The default is the current system Button appearance for windows setting. This cannot be specified here.
ICONFILE	filename	Sets the object's icon. This is equivalent to calling wpSetIconData.
ICONPOS	x, y	Sets the object's initial icon position in a folder. The x and y values represent the position in the folder in percentage coordinates.
ICONRESOURCE	id, module	Sets the object's icon. This is equivalent to calling wpSetIconData. The "id" is the icon resource ID in the dynamic link library (DLL) "module".
MINWIN	HIDE	Hides views of object, when the minimize button is selected.
	VIEWER	Minimizes views of the object to Minimize window to viewer, when the minimize button is selected.
	DESKTOP	Minimizes views of the object to Minimize window to Desktop,

		when the minimize button is selected. The default is the current Minimize button setting. This cannot be specified here.
NOCOPY	YES	Users cannot copy objects. This is equivalent to calling wpSetStyle with the OBJSTYLE_NOCOPY style.
	NO	Users can copy objects. This is the default value.
NODELETE	YES	Users cannot delete objects. This is equivalent to calling wpSetStyle with the OBJSTYLE_NODELETE style.
	NO	Users can delete objects. This is the default value.
NODRAG	YES	Users cannot drag objects. This is equivalent to calling wpSetStyle with the OBJSTYLE_NODRAG style.
	NO	Users can drag objects. This is the default value.
NODROP	YES	No other objects can be dropped on this object. This is equivalent to calling wpSetStyle with the OBJSTYLE_NODROPON style.
	NO	Objects can be dropped on this object. This is the default value.
NOLINK	YES	Objects cannot be linked. This is equivalent to calling wpSetStyle with the OBJSTYLE_NOLINK style.
	NO	Objects can be linked. This is the default value.
NOMOVE	YES	Users cannot move objects. This is equivalent to calling wpSetStyle with the OBJSTYLE_NOMOVE style.
	NO	Users can move objects. This is the default value.
NOPRINT	YES	Users cannot print objects. This is equivalent to calling wpSetStyle with the OBJSTYLE_NOPRINT style.
	NO	Users can print objects. This is the default value.
NORENAME	YES	Users cannot rename objects. This is equivalent to calling wpSetStyle with the OBJSTYLE_NORENAME style.
	NO	Users can rename objects. This is the default value.
NOSETTINGS	YES	Users cannot open the object's Settings notebook. This is equivalent to calling wpSetStyle with the OBJSTYLE_NOSETTINGS style.
	NO	Users can open the object's Settings notebook. This is the default value.

NOSHADOW (Same as NOLINK)	YES	Users cannot create shadows of objects. This is equivalent to calling wpSetStyle with the OBJSTYLE_NOLINK style.
	NO	Users can create shadows of objects. This is the default value.
NOTVISIBLE	YES	Objects are not visible. This is equivalent to calling wpSetStyle with the OBJSTYLE_NOTVISIBLE style.
	NO	Objects are visible. This is the default value.
OBJECTID	<name>	Sets a persistent ID for the object. The OBJECTID can be used to obtain a pointer or handle to the object by calling wpclsQueryObject or WinQueryObject. An OBJECTID is any unique string preceded with a < and terminated with a >. The default is to not have an ID.
OPEN	SETTINGS	Opens settings view when object is created or when WinSetObjectData is called.
	DEFAULT	Opens the default view when an object is created or when WinSetObjectData is called. The default is to not open a view.
TEMPLATE	YES	Users can create object template. This is equivalent to calling wpSetStyle with the OBJSTYLE_TEMPLATE style.
	NO	Users cannot create object template. This is the default value.
TITLE	Title	Sets the object's title. This is equivalent to calling wpSetTitle.

The following code fragments is an example of WPObject setup string:

```
PSZ pszSetupString = "TITLE    = MYObject;"
                     "ICONFILE = MYOBJ.ICO;"
                     "OBJECTID = <MyObjectID>";
```

Using Setup/Cleanup Methods

This section describes how the setup and cleanup methods relate to one another using sample code fragments.

Note: The sample code fragments in this section are part of a complete program that provides a new object class, Stack, whose instances implement standard programming push down stacks. The program is illustrated in Sample Code for Setup/Cleanup Methods.

The stack is implemented as a linked list of entries with the head of the list being the top of the stack. The following code fragments shows the content of the stack elements and the stack instance data:

```
typedef struct _STACKENTRY
{
  PSTACKENTRY pNext;
  BYTE        ReserveZeros[8];
  ULONG       cbEntry;
  BYTE        Entry[1];
} STACKENTRY *PSTACKENTRY;

BOOL        bInitialized;   // Initialization flag
PSTACKENTRY pStackTop;      // Head of stack linked list
HMTX        hmtxStack;      // Stack serialization semaphore
```

Initializing the Object's Instance Data

The wpInitData method initializes the Stack's instance data. Zero initialization is statisfactory for the stack top pointer (pStackTop), so there is nothing to do with respect to the instance data. However, I will take this opportunity to increment the awake stack instance count being maintained by the class because I want to decrement it in wpUnInitData. There are three additional items to note when overriding wpInitData:

1. The wpInitData method is invoked prior to the determination or restoration of an object's state. It is therefore necessary to be extremely careful about what other instance methods are called. It is best to call none unless you wrote them.

2. It is safest to call the parent's wpInitData before doing your own initialization.

3. If it is possible for this class to be a common ancestor in a multiple inheritance scenario, then this method needs to be written such that multiple invocations are handled. For example, if a class named Stack23 was derived from classes Stack2 and Stack3 which in turn were subclasses of Stack, that situation would exist. Not all Workplace Shell classes (WP*) currently take this precaution. Therefore, it is advisable to not inherit from more than one Workplace Shell class. Using SOM initializers and destructors is an alternative to overriding wpInitData in the multiple inheritance situation.

The following sample code shows the use of wpInitData:

```
SOM_Scope void SOMLINK stk_wpInitData(
                       Stack *somSelf) // In  - pointer to the object
{
  StackData *somThis = StackGetData(somSelf);
  StackMethodDebug("Stack","stk_wpInitData");

  parent_wpInitData(somSelf);
  if (!(_bInitialized))
  {
     _bInitialized = TRUE;
     _clsIncObjectCount(_somGetClass(somSelf), STK_AWAKECOUNT);
   } /* Endif */
   return;

}  // End stk_wpInitData
```

Initializing a Newly Created Object

The wpSetupOnce method is called once during the creation of an object. I will push one item onto the stack. Unless popped by the setup string, every newly created stack will have this item. The following sample code shows the use of wpSetupOnce:

```
SOM_Scope BOOL SOMLINK stk_wpSetupOnce(
                       Stack *somSelf,      // In  - pointer to the object
                       PSZ pszSetupString)

// The method returns the value:
//     TRUE  = successful
```

```
//     FALSE = error
{
  PSZ pszStackItem = "***** BOTTOM OF STACK *****";

  // StackData *somThis = StackGetData(somSelf);
  StackMethodDebug("Stack","stk_wpSetupOnce");

  _Push(somSelf, pszStackItem, strlen(pszStackItem) + 1);
  return (parent_wpSetupOnce(somSelf, pszSetupString));

} // End stk_wpSetupOnce
```

Setting the Object's Characteristics and Behaviors

Invoking wpSetupOnce parent causes wpSetup to be called. Two new keywords, PUSHITEM and POPITEM, have been defined for this class to push and pop items from the stack. This method is invoked when an object is created and when WinSetObjectData is called. Note the use of wpIsObjectInitialized and wpSaveDeferred to write the persistent image to the .INI file, if called as a result of WinSetObjectData. The wpScanSetupString method searches a setup string for a given keyname and returns its corresponding value. It also has a feature which returns just the length of the value when passed NULL as a pointer to the value buffer. Keynames and their values are separated by semicolons in the setup string. The escape character (^) followed by a semicolon can be used to represent a semicolon, if one is required in a keyname value specification. The following code fragments shows an example of a setup string for a stack object:

```
PSZ pszSetupString = "TITLE    = MyStack;"
                     "OBJECTID = <MyStack>;"
                     "PUSHITEM = Pushed by Setup"
```

The following sample code shows the use of wpSetup:

```
SOM_Scope BOOL SOMLINK stk_wpSetup(
                       Stack *somSelf,    // In - pointer to the object
                       PSZ   pszSetupString)

// The method returns the value:
//     TRUE  = successful
//     FALSE = error
{
  BOOL  bSaveObject = FALSE;
  BOOL  bStatus;
  ULONG cbValue;
  PSZ   pszValue;

  // StackData *somThis = StackGetData(somSelf);
  StackMethodDebug("Stack","stk_wpSetup");

  bStatus = parent_wpSetup(somSelf, pszSetupString);
  if (bStatus && pszSetupString && *pszSetupString)
  { //
    // Process PUSHITEM
    //
    if (_wpScanSetupString(somSelf,
                           pszSetupString,
                           "PUSHITEM",
                           NULL,
                           &cbValue))
    {

      pszValue = (PSZ)_wpAllocMem(somSelf, cbValue, NULL);
      bStatus  = FALSE;
      if (pszValue)
      {
        bStatus = _wpScanSetupString(somSelf,
                                     pszSetupString,
                                     "PUSHITEM",
                                     pszValue,
                                     &cbValue);

        if (bStatus)
        {
```

```
          bStatus     = _Push(somSelf, pszValue, cbValue);
          bSaveObject = bStatus;
        } /* Endif */
        _wpFreeMem(somSelf, (PBYTE)pszValue);
      } /* Endif */
    } /* Endif */

    //
    // Process POPITEM
    //
    if (bStatus && _wpScanSetupString(somSelf,
                                      pszSetupString,
                                      "POPITEM",
                                      NULL,
                                      &cbValue))
    {
      bStatus = FALSE;

      if (_Pop(somSelf, NULL, &cbValue))
      {
        pszValue = (PSZ)_wpAllocMem(somSelf, cbValue, NULL);
        if (pszValue)
        {
          bStatus     = _Pop(somSelf, pszValue, &cbValue);
          bSaveObject = bStatus;
          _wpFreeMem(somSelf, (PBYTE)pszValue);
        } /* Endif */
      } /* Endif */
    } /* Endif */

    if (bSaveObject && _wpIsObjectInitialized(somSelf))
    { //
      // Save the object to the INI file
      //
      _wpSaveDeferred(somSelf);
    } /* Endif */
  } /* Endif */
  return(bStatus);
} // End stk_wpSetup
```

Notifying the Object Is Ready

The invocation of wpObjectReady is a signal that the object creation or awakening is complete. The following sample code uses this method to increment the persistent object count:

```
SOM_Scope void SOMLINK stk_wpObjectReady(
          Stack *somSelf,       // In - pointer to the object
          ULONG ulCode,         // In - type of operation completed
          WPObject *refObject)  // In - pointer to the source
                                //      object, if copy/shadow create

{
  // StackData *somThis = StackGetData(somSelf);
  StackMethodDebug("Stack","stk_wpObjectReady");

  parent_wpObjectReady(somSelf, ulCode, refObject);
  if (!(ulCode & OR_AWAKE))
  {
     _clsIncObjectCount(_somGetClass(somSelf), STK_PERSISTENTCOUNT);
  } /* Endif */
  return;
} // End stk_wpObjectReady
```

Uninitializing the Object

The wpUnInitData method is executed just prior to deallocation of the object's memory during the process of making the object dormant or its permanent destruction. We want to deallocate all the memory and other resources allocated during this instantiation of the object. Memory allocated via wpAllocMem() is freed automatically by one of the Workplace Shell ancestors, so we need not do it here. As in the override of wpUnInitData, the multiple inheritance issue should be addressed here as well. The following sample code shows the use of wpUnInitData:

```
SOM_Scope void SOMLINK stk_wpUnInitData(
                       Stack *somSelf)  // In - pointer to the object
{
  StackData *somThis = StackGetData(somSelf);
  StackMethodDebug("Stack","stk_wpUnInitData");

  if (_bInitialized)
  {
    _bInitialized = FALSE;
    _clsDecObjectCount(_somGetClass(somSelf), STK_AWAKECOUNT);
  } /* Endif */
  parent_wpUnInitData(somSelf);
} // End stk_wpUnInitData
```

Destroying the Object

The wpFree method is generally overridden by storage classes such as WPFileSystem and WPAbstract, which provide for the permanent storage of an object's instance data. In addition, if you have an application that needs to destroy an object, call wpDelete rather than wpFree since it provides the user a chance to confirm the deletion. In this example, wpFree is overridden so that a count of stack object instances can be maintained. It is important that the somSelf pointer for the object not be used again following the call to parent_wpFree. The following sample code shows the use of wpFree:

```
SOM_Scope BOOL SOMLINK stk_wpFree(
                       Stack *somSelf)  // In - pointer to the object

// The method returns the value
//     TRUE  = successful
//     FALSE = error
{
  BOOL      bStatus;
  SOMClass  *Class;

  // StackData *somThis = StackGetData(somSelf);
  StackMethodDebug("Stack","stk_wpFree");

  Class = _somGetClass(somSelf);
  bStatus = parent_wpFree(somSelf);

  if (bStatus)
  {
    _clsDecObjectCount(Class, STK_PERSISTENTCOUNT);
  } /* Endif */
  return(bStatus);
}  // End stk_wpFree
```

Sample Code for Setup/Cleanup Methods

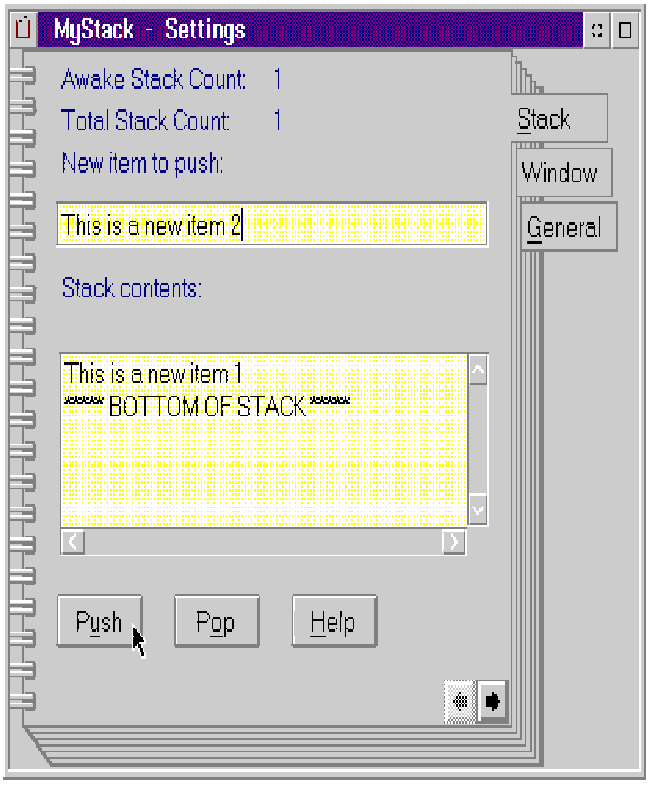
This section illustrates a complete setup/cleanup sample program.

The Workplace Shell Stack object is a sample Workplace Shell SOM application that demonstrates the use of the setup and cleanup methods. The complete program provides a new object class, Stack, whose instances implement standard programming push down stacks. WPAbstract is the parent class of the Stack class providing persistence of the stack entries, via the OS2.INI file. The Stack class also maintains the number of permanent object instances and currently awake objects (instantiated in memory).

A Settings notebook page has been added to each object allowing the user to:

- View a stack's contents
- Push new items onto the stack
- Pop existing items from the stack.

The awake and total object counts are also displayed on each object's new notebook page. The following figure shows the Stack Settings notebook:



Pushing Items onto the Stack

To push a new item onto the top of the stack, enter a text string in the New item to push field and click on the Push push button. The pushed item will appear at the top of the list.

Popping Items from the Stack

To pop the top item from the stack just click on the Pop push button. The popped item will be removed from the list and will appear in the New item to push field.

Setup/Cleanup Application Sample Code

Several parts of this program are explained in Using Setup/Cleanup Methods.

The setup/cleanup application includes the following files:

File Name	Description
WPSTACK.IDL	Class interface definition
WPSTACK.C	Source code
WPSTACK.RCH	Resource header
WPSTACK.RC	Resource definition
WPSTACK.IPF	Stack help source
WPSTACK.MAK	Stack make file for building the application.

Class Definition File for Stack

The following sample illustrates the class interface definition file (IDL):

```
//# CLASS: Stack
//#
//# CLASS HIERARCHY:
//#
//#    SOMObject
//#          WPObject
//#                WPAbstract
//#                      Stack
//#

#ifndef wpstack_idl
#define wpstack_idl

#include "wpabs.idl"
typedef somToken PSTACKENTRY;  //# Dummy typedef for instance variable

interface M_Stack;

interface Stack : WPAbstract
{
  ULONG InsertStackPage(in HWND hwndDlg);
  //# PARAMETERS:
  //#   HWND  hwndDlg   // In -  notebook dialog window handle
  //#   ULONG return    // Out - 0 = error occurred, else page #
  //#
  //# DESC: Adds the new Settings notebook page.

  BOOL Lock();
  //# PARAMETERS:
  //#   BOOL  return  // Out - TRUE  = successful
  //#                           FALSE = errror
  //#
  //# DESC: Lock the stack using a mutex semaphore so we don't have
  //#       to worry about multiple threads messing up the linked list.

  BOOL Pop(in PVOID pBuffer, inout ULONG pcbBuffer);
  //# PARAMETERS:
  //#   PVOID pBuffer,      // In    - pointer to buffer for popped item
  //#   PULONG pcbBuffer);  // InOut - size in bytes of pop buffer (in),
  //#                       //         size actually returned (out)
  //#   BOOL  return        // Out   - TRUE  = successful
  //#                                 FALSE = errror
  //# DESC: Pops an entry from the stack.

  BOOL Push(in PVOID pNewItem, in ULONG cbNewItem);
  //# PARAMETERS:
  //#   PVOID pNewItem,   // In  - pointer to the item to push
  //#   ULONG cbNewItem   // In  - size in bytes of item to push
  //#   BOOL  return      // Out - TRUE  = successful
  //#                              FALSE = errror
```

```
  //# DESC: Pushes an entry onto the stack.

  BOOL Unlock();
  //# PARAMETERS:
  //#   BOOL  return  // Out - TRUE  = successful
  //#                         FALSE = errror
  //#
  //# DESC: Unlock the stack, releasing the mutex semaphore.

  #ifdef __SOMIDL__
  implementation
  {
    releaseorder: InsertStackPage, Lock, Pop, Push, Unlock;

    /*
     * Class modifiers
     */
    functionprefix = stk_;
    majorversion   = 1;
    minorversion   = 2;
    filestem       = wpstack;
    metaclass      = M_Stack;
    callstyle      = oidl;
    dllname        = "wpstack.dll";

    /*
     * Internal instance variables
     */
    BOOL        bInitialized;
    PSTACKENTRY pStackTop;
    HMTX        hmtxStack;

    /*
     * Passthru to the .IH file the following:
     */
    passthru C_ih =
    "#define STK_APPLICATION        \"wpstack class\""
    "#define STK_PERSISTENTCOUNTKEY \"object count\""
    "#define MPNULL                 (MPFROMP( NULL ))"
    "#define MPZERO                 (MPFROMSHORT( 0 ))"
    "#define MRTRUE                 (MRFROMSHORT( (SHORT) TRUE ))"
    "#define MRFALSE                (MRFROMSHORT( (SHORT) FALSE ))"
    ""
    "typedef struct _STACKENTRY"
    "{"
    "   struct _STACKENTRY *Next;"
    "   BYTE        ReserveZeros[8];"
    "   ULONG       cbEntry;"
    "   BYTE        Entry[1];"
    "} STACKENTRY, *PSTACKENTRY;"
    "";

    /*
     * Passthru to the .H file the following:
     */
    passthru C_h =
    "#define STK_AWAKECOUNT      1"
    "#define STK_PERSISTENTCOUNT 2"
    "";

    /*
     * Method modifiers
     */
    wpAddSettingsPages:              override;
    wpFree:                          override;
    wpInitData:                      override;
    wpObjectReady:                   override;
    wpRestoreState:                  override;
    wpSaveState:                     override;
    wpSetup:                         override;
    wpSetupOnce:                     override;
    wpUnInitData:                    override;
  };
  #Endif /* __SOMIDL__ */
};

interface M_Stack : M_WPAbstract
{
  ULONG clsDecObjectCount(in ULONG ulCountType);
  //# PARAMETERS:
  //#   ULONG ulCountType  // In  - STK_AWAKECOUNT,
  //#                               STK_PERSISTENTCOUNT
  //#   ULONG return       // Out - new count value
```

```
  //#
  //# DESC: Decrements the number of defined object instances.

  ULONG clsIncObjectCount(in ULONG ulCountType);
  //# PARAMETERS:
  //#   ULONG ulCountType  // In  - STK_AWAKECOUNT,
  //#                              STK_PERSISTENTCOUNT
  //#   ULONG return       // Out - new count value
  //#
  //# DESC: Increments the number of defined object instances.

  HMODULE clsQueryModuleHandle();
  //# PARAMETERS: HMODULE return // Out - DLL module handle
  //#
  //# DESC: Gets resource module handle.

  ULONG clsQueryObjectCount(in ULONG ulCountType);
  //# PARAMETERS:
  //#   ULONG ulCountType  // In  - STK_AWAKECOUNT,
  //#                              STK_PERSISTENTCOUNT
  //#   ULONG return       // Out - count of awake/persistent instances
  //#
  //# DESC: Returns the current number of defined object instances.

  #ifdef __SOMIDL__
  implementation
  {
    releaseorder: clsDecObjectCount,
                  clsIncObjectCount,
                  clsQueryModuleHandle,
                  clsQueryObjectCount;

    /*
     * Class modifiers
     */
    functionprefix = stkM_;
    majorversion   = 1;
    minorversion   = 2;
    filestem       = wpstack;
    callstyle      = oidl;
    dllname        = "wpstack.dll";

    /*
     * Internal instance variables
     */
    HMODULE  hmod;              /* Resource module handle            */
    ULONG    ulAwakeCount;      /* No. of stack instances in memory  */
    ULONG    ulPersistentCount; /* Total number of stack instances   */

    /*
     * Method modifiers
     */
    wpclsInitData:                  override;
    wpclsQueryDefaultHelp:          override;
    wpclsQueryDefaultView:          override;
    wpclsQueryIconData:             override;
  };
  #Endif /* __SOMIDL__ */
};
#Endif  /* wpstack_idl */
```

Source Code for Stack

The following sample illustrates the source code (C):

```
// Description...... M_Stack, Stack
//
//     SOMObject
//           WPObject
```

```
//                WPAbstract
//                     Stack
//
// This class implements a persistent stack whose data is saved in the
// OS2.INI file by inheritance from WPAbstract.
//
// Instance Methods.........
//
//                  InsertStackPage            introduced
//                  Lock                       introduced
//                  Pop                        introduced
//                  Push                       introduced
//                  Unlock                     introduced
//                  wpAddSettingsPages         override
//                  wpFree                     override
//                  wpInitData                 override
//                  wpObjectReady              override
//                  wpRestoreState             override
//                  wpSaveState                override
//                  wpSetup                    override
//                  wpSetupOnce                override
//                  wpUnInitData               override
//
// Class Methods.........
//
//                  clsDecObjectCount          introduced
//                  clsIncObjectCount          introduced
//                  clsQueryModuleHandle       introduced
//                  clsQueryObjectCount        introduced
//                  wpclsInitData              override
//                  wpclsQueryDefaultHelp      override
//                  wpclsQueryDefaultView      override
//                  wpclsQueryIconData         override
//
// Non-Method Functions.......
//
//                  DialogProc
//
// ******************************************************************

#ifndef SOM_Module_wpstack_Source
#define SOM_Module_wpstack_Source
#endif
#define Stack_Class_Source
#define M_Stack_Class_Source

#include "stdlib.h"
#include "string.h"
#define  INCL_PM
#define  INCL_DOS
#define  INCL_DOSERRORS
#define  INCL_DEV
#define  INCL_WPCLASS
#define  INCL_WINWORKPLACE
#include "os2.h"
#include "wpstack.rch"
#include "wpstack.ih"

//
// Forward procedure declarations
//
MRESULT EXPENTRY DialogProc(HWND, ULONG, MPARAM, MPARAM);

//
// Global variables
//
PSZ vpszClassName   = "Stack";
PSZ vpszHelpLibrary = "wpstack.hlp";

// ******************************************************************
//
// METHOD NAME: stk_InsertStackPage
//
// FUNCTION: Inserts the new page into the Settings notebook.
//           This method exists so that subclassers can remove
//           or replace this page.
//
//******************************************************************
SOM_Scope ULONG SOMLINK stk_InsertStackPage(
   Stack *somSelf,  // In - pointer to the object
   HWND hwndDlg)    // In - notebook window handle

// Method return value out - 0 = error, otherwise new page ID
```

```
{
   PAGEINFO  pi;
   // StackData *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_InsertStackPage");

   memset((PCH) &pi, 0, sizeof(PAGEINFO));
   pi.cb                 = sizeof(PAGEINFO);
   pi.hwndPage           = NULLHANDLE;
   pi.usPageStyleFlags   = BKA_MAJOR;
   pi.usPageInsertFlags  = BKA_FIRST;
   pi.pfnwp              = DialogProc;
   pi.resid              = _clsQueryModuleHandle(_Stack);
   pi.dlgid              = DLG_STYLE;
   pi.pszName            = "~Stack";
   pi.pCreateParams      = somSelf;
   pi.pszHelpLibraryName  = vpszHelpLibrary;
   return(_wpInsertSettingsPage(somSelf, hwndDlg, &pi));

}  // End stk_InsertStackPage

// ******************************************************************
//
// METHOD NAME: stk_Lock
//
// FUNCTION: Locks the stack using a mutex semaphore so it does not
//           have to worry about multiple threads messing up the
//           linked list.
//
//******************************************************************
SOM_Scope BOOL SOMLINK stk_Lock(
   Stack *somSelf) // In - pointer to the object

// Method return value out - TRUE = successful, False = error
{
   ULONG rc;
   QMSG   qmsg;
   MQINFO mqinfo;
   StackData *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_Lock");

   if (!_hmtxStack)
   {
      rc = DosCreateMutexSem(NULL, &_hmtxStack, 0, TRUE);
   }
   else
   {
      rc = DosRequestMutexSem(_hmtxStack,100);
      if (rc)
      {

         if (WinQueryQueueInfo(HMQ_CURRENT,&mqinfo,sizeof(MQINFO)))
         {  //
            // For a PM process/thread, do a fake WinMsgSemWait
            //
            while (rc)
            {
               if (rc==ERROR_TIMEOUT)
               {
                  WinPeekMsg(WinQueryAnchorBlock(HWND_DESKTOP),
                             &qmsg,
                             NULLHANDLE,
                             0,0,
                             PM_NOREMOVE);  /* For example,         */
                                            /* fake a WinMsgSemWait */
               } /* Endthen */

               else
               {
                  break;
               } /* Endif */
               rc = DosRequestMutexSem(_hmtxStack, 100);
            } /* Endwhile */
         } /* Endthen */

         else
         {  //
            // For a non PM process/thread, just block on the semaphore
            //
            rc = DosRequestMutexSem(_hmtxStack, SEM_INDEFINITE_WAIT);
         } /* Endif */
      } /* Endif */
   } /* Endif */
   return(!rc);
```

```
}  // End stk_Lock

// ******************************************************************
//
// METHOD NAME: stk_Pop
//
// FUNCTION:  Pops an item from the stack.
//
//******************************************************************
SOM_Scope BOOL SOMLINK stk_Pop(
   Stack  *somSelf,   // In    - pointer to the object
   PVOID  pBuffer,    // In    - pointer to the buffer for popped item.
                      //         NULL means return the buffer's size
   PULONG pcbBuffer)  // InOut - size in bytes of the pop buffer (in),
                      //         size actually returned (out)

// Method return value out - TRUE = successful, FALSE = error
{
   BOOL        bSem;
   PSTACKENTRY pNextEntry;
   BOOL        bStatus = FALSE;
   StackData *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_Pop");

   /*
    * pStackTop is an instance variable, initialized to NULL,
    * which points to the top (first) entry
    * on the stack (list).
    */
   bSem = _Lock(somSelf);  // Semaphore that protects the stack chain
   if (_pStackTop)
   {
      if (pBuffer)
      {

         if (*pcbBuffer >= _pStackTop->cbEntry)
         {
            memcpy(pBuffer, _pStackTop->Entry, _pStackTop->cbEntry);
            *pcbBuffer = _pStackTop->cbEntry;
            pNextEntry = _pStackTop->Next;
            bStatus    = _wpFreeMem(somSelf, (PBYTE)_pStackTop);
            if (bStatus)
            {
               _pStackTop = pNextEntry;
            } /* Endif */
         } /* Endif */
      } /* Endthen */

      else
      {
         *pcbBuffer = _pStackTop->cbEntry;
         bStatus    = TRUE;
      } /* Endif */
   } /* Endif */
   if (bSem) _Unlock(somSelf);
   return(bStatus);

} // End stk_Pop

// ******************************************************************
//
// METHOD NAME: stk_Push
//
// FUNCTION: Pushes an item onto the stack.
//
//******************************************************************
SOM_Scope BOOL  SOMLINK stk_Push(
   Stack *somSelf,     // In - pointer to the object
   PVOID pNewItem,     // In - pointer to the item to push
   ULONG cbNewItem)    // In - size in bytes of item to push

// Method return value out - TRUE = successful, FALSE = error
{
   BOOL        bSem;
   PSTACKENTRY pNewEntry;
   BOOL        bStatus  = FALSE;
   StackData   *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_Push");

   pNewEntry = (PSTACKENTRY)_wpAllocMem(somSelf,
                cbNewItem +
                sizeof(STACKENTRY) -
```

```
                sizeof(BYTE),
                NULL);

   if (pNewEntry)
   {
      /*
       * pStackTop is an instance variable, initialized to NULL,
       * which points to the top (first) entry
       * on the stack (list)
       */
      memset(pNewEntry, 0, sizeof(STACKENTRY));
      bSem = _Lock(somSelf);  // Semaphore protect the stack chain.
      pNewEntry->Next = _pStackTop;
      _pStackTop = pNewEntry;

      if (pNewItem && cbNewItem)
      {
         pNewEntry->cbEntry = cbNewItem;
         memcpy(pNewEntry->Entry, pNewItem, cbNewItem);
      } /* Endif */
      if (bSem) _Unlock(somSelf);
      bStatus = TRUE;
   } /* Endif */
   return(bStatus);
} /* End stk_Push */

// ********************************************************************
//
// METHOD NAME: stk_Unlock
//
// FUNCTION: Unlocks the stack, releasing the mutex semaphore.
//
//********************************************************************
SOM_Scope BOOL SOMLINK stk_Unlock(
   Stack *somSelf) // In - pointer to the object

// Method return value out - TRUE = successful, False = error.
{
   ULONG rc = 0;
   StackData *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_Unlock");

   if (_hmtxStack)
   {
      rc = DosReleaseMutexSem(_hmtxStack);
   } /* Endif */
   return(!rc);

}  // End stk_Unlock

// ********************************************************************
//
// METHOD NAME: stk_wpAddSettingsPages
//
// FUNCTION:
//
//   Adds the new stack page by calling InsertStackPage().  This page
//   shows the contents of the stack and allows pushing and popping
//   entries.  It is added on top of existing pages.
//
//********************************************************************

SOM_Scope BOOL SOMLINK stk_wpAddSettingsPages(
   Stack *somSelf,    // In - pointer to the object
   HWND hwndNotebook) // In - Settings notebook window handle

// Method return value out - TRUE = successful, FALSE = error
{
    // StackData *somThis = StackGetData(somSelf);
    StackMethodDebug("Stack","stk_wpAddSettingsPages");

    parent_wpAddSettingsPages(somSelf, hwndNotebook);
    return(_InsertStackPage(somSelf, hwndNotebook) != 0);

} // End stk_wpAddSettingsPages

// ********************************************************************
//
// METHOD NAME: stk_wpFree
//
// FUNCTION:
//
//   wpFree is generally overridden by storage classes such as
```

```
//   WPFileSystem and WPAbstract, which provide for the permanent
//   storage of an object's instance data.  In addition, if you
//   are trying to destroy an object, call wpDelete rather than wpFree
//   since it provides a user confirmation capability.
//
//   In this example, wpFree is overridden so that a count of stack
//   object instances can be maintained.  It is important that the
//   somSelf pointer for the object not be used again following the
//   call to parent_wpFree.
//
//*****************************************************************
SOM_Scope BOOL SOMLINK stk_wpFree(
   Stack *somSelf)  // In - pointer to the object

// Method return value out - TRUE = successful, FALSE = error
{
   BOOL      bStatus;
   SOMClass  *Class;
   // StackData *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_wpFree");

   Class   = _somGetClass(somSelf);
   bStatus = parent_wpFree(somSelf);
   if (bStatus)
   {
      _clsDecObjectCount(Class, STK_PERSISTENTCOUNT);
   } /* Endif */
   return(bStatus);

}  // End stk_wpFree

// ******************************************************************
//
// METHOD NAME:  stk_wpInitData
//
// FUNCTION:
//
//   Initializes the stack's instance data.  Zero initialization is
//   statisfactory for the stack top pointer (pStackTop), so there is
//   nothing to do with respect to the instance data.  However, I will take
//   this opportunity to increment the awake stack instance count being
//   maintained by the class because I want to decrement it in wpUnInitData.
//   There are three additional items to note when overriding wpInitData.

//   First, wpInitData is invoked prior to the determination or restoration
//   of an object's state.  It is therefore necessary to be extremely
//   careful about what other instance methods are called.  It is best to
//   call none unless you wrote them.
//
//   Second, it is safest to call the parent's wpInitData method before
//   doing your own initialization.
//
//   Third, if it is possible for this class to be a common ancestor in a
//   multiple inheritance scenario, then this method needs to be written
//   such that multiple invocations are handled.  For example, if a class
//   named Stack23 was derived from classes Stack2 and Stack3 which in
//   turn were subclasses of Stack, that situation would exist.  Not
//   all Workplace Shell classes (WP*.)  currently take this precaution.
//   Therefore, it is advisable not to inherit from more than one
//   Workplace Shell class.  Using SOM initializers and destructors is
//   an alternative to overriding wpInitData in the multiple
//   inheritance situation.
//
//*****************************************************************

SOM_Scope void SOMLINK stk_wpInitData(
   Stack *somSelf)  // In - pointer to the object
{
    StackData *somThis = StackGetData(somSelf);
    StackMethodDebug("Stack","stk_wpInitData");

    parent_wpInitData(somSelf);
    if (!(_bInitialized))
    {
       _bInitialized = TRUE;
       _clsIncObjectCount(_somGetClass(somSelf), STK_AWAKECOUNT);
    } /* Endif */
    return;

}  // End stk_wpInitData

// ******************************************************************
//
```

```
// METHOD NAME:  stk_wpObjectReady
//
// FUNCTION:
//
//   Notifies that the object's creation/awakening is complete.
//   This example uses this method to increment the persistent
//   object count.
//
//*************************************************************************

SOM_Scope void SOMLINK stk_wpObjectReady(
   Stack    *somSelf,    // In - pointer to the object
   ULONG    ulCode,      // In - type of operation completed
   WPObject *refObject)  // In - pointer to the source object
                         //      if an object's
                         //      copy/shadow is created

{
    // StackData *somThis = StackGetData(somSelf);
    StackMethodDebug("Stack","stk_wpObjectReady");

    parent_wpObjectReady(somSelf, ulCode, refObject);
    if (!(ulCode & OR_AWAKE))
    {
       _clsIncObjectCount(_somGetClass(somSelf), STK_PERSISTENTCOUNT);
    } /* Endif */
    return;

} // End stk_wpObjectReady

// ************************************************************************
//
// METHOD NAME:  stk_wpRestoreState
//
// FUNCTION:
//
//   Restores the stack items from the persistent INI file storage
//   into the memory resident linked list pointed to by the
//   pStackTop instance variable.
//
//*************************************************************************

SOM_Scope BOOL  SOMLINK stk_wpRestoreState(
   Stack *somSelf,    // In - pointer to the object
   ULONG ulReserved)

{
   ULONG       i;
   ULONG       ulItemCount;
   ULONG       cbItem;
   PBYTE       pItem;
   PSTACKENTRY pPrevItem = NULL;
   BOOL        bStatus;
   StackData *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_wpRestoreState");

   bStatus = _wpRestoreLong(somSelf, vpszClassName, 0, &ulItemCount);
   for (i=1; (i<=ulItemCount) && bStatus; i++)
   {
      bStatus = _wpRestoreData(somSelf, vpszClassName, i, NULL, &cbItem);
      if (bStatus)
      {
         pItem = _wpAllocMem(somSelf, cbItem, NULL);

         if (pItem)
         {
            bStatus = _wpRestoreData(somSelf, vpszClassName, i, pItem, &cbItem);
            if (bStatus)
            {
               if (pPrevItem)
               {
                  pPrevItem->Next = (PSTACKENTRY)pItem;
               } /* Endthen */

               else
               {
                  _pStackTop = (PSTACKENTRY)pItem;
               } /* Endif */
               pPrevItem = (PSTACKENTRY)pItem;
            } /* Endthen */

            else
            {
```

```
                _wpFreeMem(somSelf, pItem);
             } /* Endif */
          } /* Endif */
       } /* Endif */
    } /* Endfor */
    return (parent_wpRestoreState(somSelf, ulReserved));

} // End stk_wpRestoreState

// ********************************************************************
//
// METHOD NAME: stk_wpSaveState
//
// FUNCTION:
//
//   Saves the memory resident linked list to a buffer, which will be
//   written to a permanent INI file storage by the WPAbstract class.
//   The memory resident linked list is pointed to by the
//   pStackTop instance variable.
//
//********************************************************************

SOM_Scope BOOL  SOMLINK stk_wpSaveState(
   Stack *somSelf)    // In - pointer to the object

{
   BOOL        bSem;
   ULONG       i;
   ULONG       ulItemCount = 0;
   ULONG       cbItem;
   PSTACKENTRY pItem;
   BOOL        bStatus;
   StackData *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_wpSaveState");

   bSem = _Lock(somSelf);  // Semaphore that protects the stack chain
   for (pItem = _pStackTop; pItem; pItem = pItem->Next, ulItemCount++){;};
   bStatus = _wpSaveLong(somSelf, vpszClassName, 0, ulItemCount);
   for (pItem = _pStackTop, i = 1; bStatus && pItem; pItem = pItem->Next, i++)
   {
      cbItem  = sizeof(STACKENTRY) + pItem->cbEntry - 1;
      bStatus = _wpSaveData(somSelf, vpszClassName, i, (PBYTE)pItem, cbItem);
   } /* Endfor */
   if (bSem) _Unlock(somSelf);
   return(parent_wpSaveState(somSelf));

} // End stk_wpSaveState

// ********************************************************************
//
// METHOD NAME:  stk_wpSetup
//
// FUNCTION:
//
//   Two new keywords, PUSHITEM and POPITEM, have been defined for this
//   class to push and pop items from the stack.  This method is invoked
//   when an object is created and when WinSetObjectData() is called.
//   wpIsObjectInitialized and wpSaveDeferred are used to write
//   the persistent image to the INI file, if they are called as a result
//   of WinSetObjectData.
//
//********************************************************************

SOM_Scope BOOL  SOMLINK stk_wpSetup(
   Stack *somSelf,    // In - pointer to the object
   PSZ   pszSetupString)

// Method return value out - TRUE = successful, FALSE = error
{
   BOOL  bSaveObject = FALSE;
   BOOL  bStatus;
   ULONG cbValue;
   PSZ   pszValue;

   // StackData *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_wpSetup");

   bStatus = parent_wpSetup(somSelf, pszSetupString);
   if (bStatus && pszSetupString && *pszSetupString)

   {  //
      // Process PUSHITEM
      //
```

```
      if (_wpScanSetupString(somSelf,
                             pszSetupString,
                             "PUSHITEM",
                             NULL,
                             &cbValue))
      {
         pszValue = (PSZ)_wpAllocMem(somSelf, cbValue, NULL);
         bStatus  = FALSE;

         if (pszValue)
         {
            bStatus = _wpScanSetupString(somSelf,
                                         pszSetupString,
                                         "PUSHITEM",
                                         pszValue,
                                         &cbValue);

            if (bStatus)
            {
               bStatus     = _Push(somSelf, pszValue, cbValue);
               bSaveObject = bStatus;
            } /* Endif */
            _wpFreeMem(somSelf, (PBYTE)pszValue);
         } /* Endif */
      } /* Endif */

      //
      // Process POPITEM
      //
      if (bStatus &&
          _wpScanSetupString(somSelf,
                             pszSetupString,
                             "POPITEM",
                             NULL,
                             &cbValue))
      {
         bStatus = FALSE;

         if (_Pop(somSelf, NULL, &cbValue))
         {
            pszValue = (PSZ)_wpAllocMem(somSelf, cbValue, NULL);
            if (pszValue)
            {
               bStatus     = _Pop(somSelf, pszValue, &cbValue);
               bSaveObject =  bStatus;
               _wpFreeMem(somSelf, (PBYTE)pszValue);
            } /* Endif */
         } /* Endif */
      } /* Endif */

      if (bSaveObject && _wpIsObjectInitialized(somSelf))
      {  //
         // Save the object to the INI file.
         //
         _wpSaveDeferred(somSelf);
      } /* Endif */
   } /* Endif */
   return(bStatus);

} // End stk_wpSetup

// ******************************************************************
//
// METHOD NAME:  stk_wpSetupOnce
//
// FUNCTION:
//
//   This method is called once during the creation of an object.
//   One item is pushed onto the stack.  Unless popped by the
//   setup string, every newly created stack will have this item.
//
//******************************************************************

SOM_Scope BOOL  SOMLINK stk_wpSetupOnce(
   Stack *somSelf,    // In - pointer to the object
   PSZ   pszSetupString)

// Method return value out - TRUE = successful, FALSE = error
{
    PSZ pszStackItem = "***** BOTTOM OF STACK *****";
    // StackData *somThis = StackGetData(somSelf);
    StackMethodDebug("Stack","stk_wpSetupOnce");
```

```
    _Push(somSelf, pszStackItem, strlen(pszStackItem) + 1);
    return (parent_wpSetupOnce(somSelf, pszSetupString));

} // End stk_wpSetupOnce

// ******************************************************************
//
// METHOD NAME:  stk_wpUnInitData
//
// FUNCTION:
//
//    This method is executed just prior to the deallocation of the
//    object's memory, during the process of making the object dormant
//    or its permanent destruction.  All memory and other resources allocated
//    during the instantiation of the object will be deallocated.  Memory
//    allocated by wpAllocMem() will be freed automatically by one of the
//    Workplace Shell ancestors, so there is no need to do this here.  As in
//    the wpInitData() override, the multiple inheritance issue should be
//    addressed here as well.
//
//******************************************************************

SOM_Scope void  SOMLINK stk_wpUnInitData(
   Stack *somSelf)    // In - pointer to the object
{
   StackData *somThis = StackGetData(somSelf);
   StackMethodDebug("Stack","stk_wpUnInitData");

   if (_bInitialized)
   {
      _bInitialized = FALSE;
      _clsDecObjectCount(_somGetClass(somSelf), STK_AWAKECOUNT);
   } /* Endif */
   parent_wpUnInitData(somSelf);

} // End stk_wpUnInitData

// ******************************************************************
//
// METHOD NAME:  stkM_clsDecObjectCount
//
// FUNCTION:
//
//    Decrements the "awake" or "persistent" object instance count
//    for the class.
//
//******************************************************************

SOM_Scope ULONG SOMLINK stkM_clsDecObjectCount(
   M_Stack *somSelf,  // In - pointer to the class object
   ULONG ulCountType) // In - STK_AWAKECOUNT,
                      //      STK_PERSISTENTCOUNT

// Method return value out - new count value
{
   ULONG ulCount = 0;

   M_StackData *somThis = M_StackGetData(somSelf);
   M_StackMethodDebug("M_Stack","stkM_clsDecObjectCount");

   switch (ulCountType)
   {
      case STK_AWAKECOUNT:
         ulCount = --_ulAwakeCount;
         break;

      case STK_PERSISTENTCOUNT:
         ulCount = --_ulPersistentCount;
         //
         // Write this value to the INI file
         //
         PrfWriteProfileData(HINI_USERPROFILE,
                             STK_APPLICATION,
                             STK_PERSISTENTCOUNTKEY,
                             &_ulPersistentCount,
                             sizeof(ULONG));
         break;

      default:
         break;
   } /* Endswitch */
   return(ulCount);
```

```
}  // stkM_clsDecObjectCount

// ************************************************************************
//
// METHOD NAME:  stkM_clsIncObjectCount
//
// FUNCTION:
//
//    Increments the "awake" or "persistent" object instance count
//    for the class.
//
//************************************************************************

SOM_Scope ULONG somlink stkM_clsIncObjectCount(
   M_Stack *somSelf,  // In - pointer to the class object
   ULONG ulCountType) // In - STK_AWAKECOUNT,
                      //      STK_PERSISTENTCOUNT

// Method return value out - new count value
{
   ULONG ulCount = 0;

   M_StackData *somThis = M_StackGetData(somSelf);
   M_StackMethodDebug("M_Stack","stkM_clsIncObjectCount");

   switch (ulCountType)
   {
      case STK_AWAKECOUNT:
         ulCount = ++_ulAwakeCount;
         break;

      case STK_PERSISTENTCOUNT:
         ulCount = ++_ulPersistentCount;
         //
         // Need to write this value to the INI file
         //
         PrfWriteProfileData(HINI_USERPROFILE,
                             STK_APPLICATION,
                             STK_PERSISTENTCOUNTKEY,
                             &_ulPersistentCount,
                             sizeof(ULONG));
         break;

      default:
         break;
   } /* Endswitch */
   return(ulCount);

} // stkM_clsIncObjectCount

// ************************************************************************
//
// METHOD NAME:  stkM_clsQueryModuleHandle
//
// FUNCTION:
//
//    Returns the resource file module handle which was obtained
//    by wpclsInitData during the initialization process.
//
//************************************************************************

SOM_Scope HMODULE somlink stkM_clsQueryModuleHandle(
   M_Stack *somSelf)  // In - pointer to the class object

// Method return value out - resouce module handle
{
   M_StackData *somThis = M_StackGetData(somSelf);
   M_StackMethodDebug("M_Stack","stkM_clsQueryModuleHandle");

   return(_hmod);

} // End stkM_clsQueryModuleHandle

// ************************************************************************
//
// METHOD NAME:  stkM_clsQueryObjectCount
//
// FUNCTION:
//
//    Returns the "awake" or "persistent" object instance count
//    for the class.
//
//************************************************************************
```

```
SOM_Scope ULONG somlink stkM_clsQueryObjectCount(
   M_Stack *somSelf,  // In - pointer to the class object
   ULONG ulCountType) // In - STK_AWAKECOUNT,
                      //      STK_PERSISTENTCOUNT

// Method return value out - current count value
{
   ULONG ulCount = 0;

   M_StackData *somThis = M_StackGetData(somSelf);
   M_StackMethodDebug("M_Stack","stkM_clsQueryObjectCount");

   switch (ulCountType)
   {
      case STK_AWAKECOUNT:
         ulCount = _ulAwakeCount;
         break;

      case STK_PERSISTENTCOUNT:
         ulCount = _ulPersistentCount;
         break;

      default:
         break;
   } /* Endswitch */
   return(ulCount);

} // End stkM_clsQueryObjectCount

// ********************************************************************
//
// METHOD NAME:  stkM_wpclsInitData
//
// FUNCTION:
//
//    Initializes the class data (resource file module handle).
//    In this case, the resouce file is linked with the code
//    (WPSTACK.DLL).  The parent initialization is usually done first so
//    that some ancestor methods can be called.
//
//********************************************************************

SOM_Scope void somlink stkM_wpclsInitData(
   M_Stack *somSelf)  // In - pointer to the class object
{
   PSZ         psz;
   somId       stackId;
   ULONG       cbValue  = sizeof(ULONG);
   M_StackData *somThis = M_StackGetData(somSelf);
   M_StackMethodDebug("M_Stack","stkM_wpclsInitData");

   parent_wpclsInitData(somSelf);
   stackId = somIdFromString("Stack");
   psz = _somLocateClassFile(SOMClassMgrObject,
                             stackId,
                             Stack_MajorVersion,
                             Stack_MinorVersion);
   SOMFree(stackId);

   if (psz != NULL)
   {
      DosQueryModuleHandle(psz, &_hmod);
   } /* Endif */

   //
   // Get the persistent object count from the INI file
   //
   PrfQueryProfileData(HINI_USERPROFILE,
                       STK_APPLICATION,
                       STK_PERSISTENTCOUNTKEY,
                       &_ulPersistentCount,
                       &cbValue);
   return;

} // End stkM_wpclsInitData

// ********************************************************************
//
// METHOD NAME:  stkM_wpclsQueryDefaultHelp
//
// FUNCTION:
//
```

```
//    Returns the help panel DLL and ID.
//
//*****************************************************************

SOM_Scope BOOL somlink stkM_wpclsQueryDefaultHelp(
   M_Stack *somSelf,       // In - pointer to the class object
   PULONG  pHelpPanelId,
   PSZ     pszHelpLibrary)

// Method return value out - TRUE = successful, FALSE = error
{
    // M_StackData *somThis = M_StackGetData(somSelf);
    M_StackMethodDebug("M_Stack","stkM_wpclsQueryDefaultHelp");

    if (pHelpPanelId)
    {
       *pHelpPanelId = ID_HELP_STACK;
    } /* Endif */

    if (pszHelpLibrary)
    {
       strcpy(pszHelpLibrary, vpszHelpLibrary);
    } /* Endif */
    return(TRUE);

} // End stkM_wpclsQueryDefaultHelp

// ******************************************************************
//
// METHOD NAME:  stkM_wpclsQueryDefaultView
//
// FUNCTION:
//
//    Opens the Settings view as the default.
//
//*****************************************************************

SOM_Scope ULONG somlink stkM_wpclsQueryDefaultView(
   M_Stack *somSelf)  // In - pointer to the class object
{
   // M_StackData *somThis = M_StackGetData(somSelf);
   M_StackMethodDebug("M_Stack","stkM_wpclsQueryDefaultView");

   return OPEN_SETTINGS;

}  // End stkM_wpclsQueryDefaultView

// ******************************************************************
//
// METHOD NAME:  stkM_wpclsQueryIconData
//
// FUNCTION:
//
//    This method is overridden so that the class' objects has
//    a unique icon.
//
//*****************************************************************

SOM_Scope ULONG SOMLINK stkM_wpclsQueryIconData(
   M_Stack   *somSelf,   // In - pointer to the class object
   PICONINFO pIconInfo ) // In - buffer for the icon data

// Method return value out - size of data returned
{
    // M_StackData *somThis = M_StackGetData(somSelf);
    M_StackMethodDebug("M_Stack","stkM_wpclsQueryIcon");

   if (pIconInfo)
   {
      pIconInfo->fFormat = ICON_RESOURCE;
      pIconInfo->hmod    = _clsQueryModuleHandle(somSelf);
      pIconInfo->resid   = ID_OBJECTICON;
   }
   return sizeof(ICONINFO);

} // End stkM_wpclsQueryIconData

// ******************************************************************
//
// PROCEDURE NAME:  DialogProc
//
// FUNCTION:
//
```

```
//   This is the dialog procedure for the new Settings notebook page.
//
//*********************************************************************

MRESULT EXPENTRY DialogProc(HWND hwndDlg,
                            ULONG msg,
                            MPARAM mp1,
                            MPARAM mp2)

{
   typedef struct _WINDATA
   {
      SOMAny     *somSelf;
      StackData  *somThis;
   }  WINDATA, *PWINDATA;

   MRESULT     mresultWpRtnCd = MRFALSE;
   PWINDATA    pwin = (PWINDATA) WinQueryWindowPtr(hwndDlg, QWL_USER);
   CHAR        szNewItem[CCHMAXPATH];
   ULONG       cbNewItem;
   PSTACKENTRY pItem;
   BOOL        bSem;

   switch( msg )
   {
      case WM_INITDLG:
      {
         pwin = (PWINDATA) _wpAllocMem((SOMAny *) mp2,
                                       sizeof(WINDATA), NULL);
         WinSetWindowPtr(hwndDlg, QWL_USER, pwin);

         //
         // Initialize WINDATA structure
         //
         pwin->somSelf     = (SOMAny *) mp2;
         pwin->somThis     = StackGetData(pwin->somSelf);

         //
         // Display the stack object count statistics
         //
         _ltoa(_clsQueryObjectCount(_somGetClass(pwin->somSelf),
                                    STK_AWAKECOUNT),
               szNewItem, 10);
         WinSetDlgItemText(hwndDlg, DLG_AWAKECOUNT, szNewItem);
         _ltoa(_clsQueryObjectCount(_somGetClass(pwin->somSelf),
                                    STK_PERSISTENTCOUNT),
               szNewItem, 10);
         WinSetDlgItemText(hwndDlg, DLG_PERSISTENTCOUNT, szNewItem);

         //
         // Insert stack items into the list box
         //
         bSem = _Lock(pwin->somSelf);
         for (pItem = pwin->somThis->pStackTop; pItem; pItem = pItem->Next)
         {
            WinInsertLboxItem(WinWindowFromID(hwndDlg, DLG_ITEMLIST),
                              LIT_END, pItem->Entry);
         } /* Endfor */
         if (bSem) _Unlock(pwin->somSelf);

         //
         // Return TRUE to tell PM that we changed focus
         //
         mresultWpRtnCd = (MRESULT) TRUE;
         break;
      }  /* End of case WM_INITDLG */

      case WM_COMMAND:
      {
         switch(SHORT1FROMMP(mp1))
         {
            case DLG_PUSH:    /* Push pushbutton  */
            {
               szNewItem[0] = '\0';
               cbNewItem = WinQueryDlgItemText(hwndDlg,
                                               DLG_NEWITEM,
                                               sizeof(szNewItem),
                                               szNewItem);

               if (_Push(pwin->somSelf, szNewItem, cbNewItem+1))
               {
                  WinInsertLboxItem(WinWindowFromID(hwndDlg, DLG_ITEMLIST),
                                    0, szNewItem);
```

```
                  _wpSaveDeferred(pwin->somSelf);  // Save state to INI file
               } /* Endif */
               break;
            }

            case DLG_POP:   /* Pop pushbutton */
            {
               szNewItem[0] = '\0';
               cbNewItem = sizeof(szNewItem);

               if (_Pop(pwin->somSelf, szNewItem, &cbNewItem))
               {
                  cbNewItem = cbNewItem ? cbNewItem : 1;
                  WinSetDlgItemText(hwndDlg, DLG_NEWITEM, szNewItem);
                  WinDeleteLboxItem(WinWindowFromID
                                    (hwndDlg, DLG_ITEMLIST), 0);
                  _wpSaveDeferred(pwin->somSelf);  // Save state to INI file
               } /* Endif */
               break;
            }

            case DLG_HELP:    /* Help push button  */
            {
               _wpDisplayHelp(pwin->somSelf, ID_HELP_STACK, vpszHelpLibrary);
               break;
            }

            default:
               break;
         }
         break;
      }  /* End of case WM_COMMAND */

      case WM_DESTROY:
      {
         _wpFreeMem(pwin->somSelf, (PBYTE)pwin);
         mresultWpRtnCd = WinDefDlgProc(hwndDlg, msg, mp1, mp2);
         break;
      }  /* End of case WM_DESTROY */

      default:
      {
         mresultWpRtnCd = WinDefDlgProc(hwndDlg, msg, mp1, mp2);
         break;
      } // End of default:
   }
   return mresultWpRtnCd;
\}
#pragma info(nouse)
```

Resource Header File for Stack

The following sample illustrates the resource header file (RCH):

```
#define ID_OBJECTICON               255
#define ID_HELP_STACK               256

#define DLG_STYLE                   100
#define DLG_AWAKECOUNT              101
#define DLG_PERSISTENTCOUNT         102
#define DLG_NEWITEM                 103
#define DLG_ITEMLIST                104
#define DLG_PUSH                    105
#define DLG_POP                     106
#define DLG_HELP                    107
```

The following sample illustrates the resource definition file (RC):

```
#include <os2.h>
#include "wpstack.rch"

ICON  ID_OBJECTICON    WPSTACK.ICO

DLGTEMPLATE DLG_STYLE LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG  "", DLG_STYLE, 0, 0, 224, 200, NOT FS_DLGBORDER
    BEGIN
        /*
         * Create output fields for stack object count statistics.
         * Will display them on every object.
         */
        LTEXT           "Awake Stack Count:"  -1,  10, 140,  85,  8
        LTEXT           ""        DLG_AWAKECOUNT, 100, 140,  25,  8
        LTEXT           "Total Stack Count:"  -1,  10, 130,  85,  8
        LTEXT           ""   DLG_PERSISTENTCOUNT, 100, 130,  25,  8

        /*
         * Create an entry field for specifying text to push
         * and a list box for the current stack contents
         */
        LTEXT           "New item to push:"  -1,  10, 120,  76,  8
        ENTRYFIELD      "",         DLG_NEWITEM,  10, 105, 180,  8,
                        ES_MARGIN

        LTEXT           "Stack contents:",   -1,  10,  90,  76,  8
        LISTBOX                     DLG_ITEMLIST,  10,  30, 180, 54,
                        LS_HORZSCROLL

        /*
         * Align the push button along the bottom
         */
        PUSHBUTTON      "P~ush",   DLG_PUSH,        7, 7, 38, 14
        PUSHBUTTON      "P~op",    DLG_POP,        57, 7, 38, 14
        PUSHBUTTON      "~Help",   DLG_HELP,      107, 7, 38, 14
    END
END
```

Help Source File for Stack

The following sample illustrates the help source file (IPF):

```
:userdoc.
.*---------------------------------------------------------------
.*  Main window extended help panel
.*---------------------------------------------------------------
.*
:h1 res=256 name=PANEL_MAIN.Workplace Shell Stack Object
:i1 id=abtStl.Workplace Shell Stack Object
:p.
The Workplace Shell Stack object is a sample Workplace Shell SOM
application that demonstrates the use of Workplace
Shell setup and cleanup methods.
The application provides a new object class, Stack, whose
instances implement standard programming push down stacks.
WPAbstract is the parent class of the Stack class providing persistence
of the stack entries, via the OS2.INI file.
The Stack class also maintains the number of permanent
object instances and currently awaken objects
(instantiated in memory).
:p.
A Settings notebook page has been added to each object allowing the
user to view a stacks contents, push new items onto the stack, and
pop existing items from the stack.
The awake and total object counts are also displayed on each
object's new notebook page.
:p.
:hp2.Pushing Items onto the Stack:ehp2.
:p.
To push a new item onto the top of the stack, enter a text string in
the input field labeled New item to push and click on the
```

```
Push push button.
The pushed item will appear at the top of the list.
:hp2.Popping Items from the Stack:ehp2.
:p.
To pop the top item from the stack just click on the
Pop push button.
The popped item will be removed from the list and appear in
the New item to push field.
.*
:euserdoc.
```

Make File for Stack

The following sample illustrates the make file (MAK):

```
.SUFFIXES: .c .obj .dll .idl .h .ih .rc .res

SC        = sc
SCFLAGS   = -maddstar -S128000 -C128000 -mnoint -v
SCLIST    = -s "ih;h;c;def"

INCLUDE   = .;d:\som\include;d:\wpshidl\h;$(INCLUDE)
SMINCLUDE = .;d:\som\include;d:\wpshidl\idl;$(SMINCLUDE)
LIB       = d:\som\lib;e:\ibmcpp\lib;e:\toolkt21\os2lib;$(LIB)
CFLAGS    = -Ge- -Ss+ -C+ -Kb -Q+ -Ms
LFLAGS    = /MAP /NOL /NOI /EXEPACK:2 /PACKCODE /PACKDATA /FAR /ALIGN:4

b=wpstack

all:         $(b).dll $(b).ih $(b).hlp

.c.obj:       icc $(CFLAGS) $*.c

$(b).dll:    $(b).ih $(b).obj $(b).def $(b).res
             link386 $(b) $(LFLAGS),$(b).dll,$(b).map,somtk,$(b)
             rc $(b).res $(b).dll
             mapsym $(b).map

$(b).obj:    $(b).c $(b).h

$(b).res:    $(b).rc $(b).ico
             rc -r $(b).rc

$(b).ih:     $(b).idl
             $(SC) $(SCFLAGS) $(SCLIST) $(@B).idl

$(b).hlp:    $(b).ipf
             ipfc $(b).ipf

clean:       -del $(b).ih  $(b).h   $(b).obj $(b).dll
             -del $(b).map $(b).sym $(b).res $(b).def
```

Object Memory: Memory Allocation Methods

The Workplace Shell provides methods which you can override to design your own memory management system, or utilize to manage memory required for objects of your application. This chapter provides information on memory management methods for Workplace Shell applications.

About Memory Allocation Methods

The wpAllocMem method allocates memory for use by an object. A record of memory blocks allocated by calls to wpAllocMem is kept by adding USAGE_MEMORY type items to the object's USEITEM list. The USEITEM list is examined by wpUnInitData when the object is destroyed or made dormant to free previously allocated resources.

Memory that is allocated to an object is de-allocated automatically when the object is no longer in use, assuming the memory has not already been cleaned up by the object. wpFreeMem is called to free allocated memory, even if the object is still in use. This method should be overriden to free memory that was allocated with an override method of wpAllocMem.

Using Memory Allocation Methods

As an example of the way these methods may be utilized, consider the definition of a WPAbstract subclass, Stack, with the Push() and Pop() methods. The following sample code fragments show the Push() and Pop() methods using the memory allocation methods. Push() allocates space for the new item, initializes its contents and places it on the stack while Pop() removes the top item from the stack, and returns its contents. The stack is implemented as a linked list of entries with the head of the list being the top of the stack. Assume the existence of methods, Lock() and Unlock(), which protect the stack linked list in a multithread environment.

Note: The sample code fragments in this section are part of a complete program that provides a new object class, Stack, whose instances implement standard programming push down stacks. The program is illustrated in "Sample Code for Setup/Cleanup Methods".

```
#include "stack.ih"

typedef struct _STACKENTRY
  {
    PSTACKENTRY pNext;
    BYTE        ReserveZeros[8];
    ULONG       cbEntry;
    BYTE        Entry[1];
  } STACKENTRY; *PSTACKENTRY;

/***********************  wpAllocMem Method  ***********************/

SOM_Scope BOOL SOMLINK stk_Push(
          Stack *somSelf,     // In - pointer to the object
          PVOID pNewItem,     // In - pointer to the item to push
          ULONG cbNewItem)    // In - size in bytes of item to push

// The method returns the value:
//     TRUE  = successful
//     FALSE = error
{
  BOOL        bSem;
  PSTACKENTRY pNewEntry;
  BOOL        bStatus = FALSE;
  StackData  *somThis = StackGetData(somSelf);
  StackMethodDebug("Stack","stk_Push");

  pNewEntry = (PSTACKENTRY)_wpAllocMem(somSelf,
              cbNewItem +
              sizeof(STACKENTRY) -
              sizeof(BYTE),
              NULL);

  if (pNewEntry)
  {
    /*
     * pStackTop is an instance variable, initialized to NULL,
     * which points to the top (first) entry
     * on the stack (list)
     */
    memset(pNewEntry, 0, sizeof(STACKENTRY));
    bSem = _Lock(somSelf);  // Semaphore that protects the stack chain
    pNewEntry->Next = _pStackTop;
    _pStackTop = pNewEntry;

    if (pNewItem && cbNewItem)
    {
      pNewEntry->cbEntry = cbNewItem;
      memcpy(pNewEntry->Entry, pNewItem, cbNewItem);
    } /* Endif */
```

```
    if (bSem) _Unlock(somSelf);
    bStatus = TRUE;
  } /* Endif */
  return(bStatus);
} /* End stk_Push */


/************************  wpFreeMem Method  ************************/

SOM_Scope BOOL SOMLINK stk_Pop(
          Stack *somSelf,   // In    - pointer to the object
          PVOID  pBuffer,   // In    - pointer to buffer for popped item,
                            //         NULL means return size of buffer
          PULONG pcbBuffer) // InOut - size in bytes of pop buffer,
                            //         size actually returned

// The method returns the value:
//     TRUE  = successful
//     FALSE = error
{
  BOOL        bSem;
  PSTACKENTRY pNextEntry;
  BOOL        bStatus = FALSE;
  StackData  *somThis = StackGetData(somSelf);
  StackMethodDebug("Stack","stk_Pop");

  /*
   * pStackTop is an instance variable, initialized to NULL,
   * which points to the top (first) entry
   * on the stack (list).
   */
  bSem = _Lock(somSelf);  // Semaphore that protects the stack chain

  if (_pStackTop)
  {
    if (pBuffer)
    {
      if (*pcbBuffer >= _pStackTop->cbEntry)
      {
        memcpy(pBuffer, _pStackTop->Entry, _pStackTop->cbEntry);
        *pcbBuffer = _pStackTop->cbEntry;

        pNextEntry = _pStackTop->Next;
        bStatus    = _wpFreeMem(somSelf, (PBYTE)_pStackTop);
        if (bStatus)
        {
          _pStackTop = pNextEntry;
        } /* Endif */
      } /* Endif */

    } /* Endthen */
    else
    {
      *pcbBuffer = _pStackTop->cbEntry;
      bStatus    = TRUE;
    } /* Endif */
  } /* Endif */

  if (bSem) _Unlock(somSelf);
  return(bStatus);
} // End stk_Pop
```

Object Mobility: Direct Manipulation Methods

Direct manipulation is interaction with objects by way of a pointing device. For example, using direct manipulation, a user can move an object from one container to another by dragging an object icon from a source container and dropping it on a target container. Direct manipulation methods allow you to specify how objects that are directly effected by such a move communicate with each other. This chapter outlines the protocols of direct manipulation with respect to their interactions with the Workplace Shell. The following material is presented as a supplement to information on direct manipulation in the *Presentation Manager Programming Guide - Advanced Topics* .

About Direct Manipulation Methods

Direct manipulation enables the user to visually drag an object (the source object) and drop it on another object (the target object) causing an interaction, or data exchange, between the objects. The Workplace Shell tracks an object being dragged, notifying target objects and windows as the source object is being dragged over them and when it is dropped on them. The Workplace Shell notifies a target object by calling the object's direct manipulation methods that process the source object being dragged and dropped. Direct manipulation methods initialize the structures that convey information about objects being moved to the target and describe images to be displayed during the Drag operation.

Direct Manipulation on Workplace Shell Objects

Common objects are provided by the Workplace Shell for use in applications. When an icon is dragged over a Workplace Shell object, the target object sends a message to the source container to enable a direct manipulation exchange. The message explains what the target object does. Completion of a drop requires that the source and target objects share a common rendering mechanism. If the source object is not appropriate for a drop on the target and the drop is prohibited, a "do-not" symbol is shown. Workplace Shell handles standard Presentation Manager rendering mechanisms, as well as its own mechanisms, to allow easy integration of Presentation Manager applications.

Workplace Shell simplifies implementation of a direct manipulation operation, handling the filling of data structures, the drag sequence, and notification of the target when a successful drop occurs. The standard direct manipulation capabilities of the Workplace Shell include **Print**, **Delete** and **File Move/Copy**.

Direct Manipulation on PM Applications

For PM applications, the Workplace Shell sends DM_ messages to PM windows using the standard or enhanced direct manipulation protocols. The Workplace Shell will drag source objects rendered as OBJECT or as OS2FILE and will accept source objects rendered in the same way. The Workplace Shell also will send a DM_PRINTOBJECT message to items dropped on the printer object.

Users can drag source objects over windows that an object creates. When this occurs, the Workplace Shell sends DM_ messages to these windows. Therefore, window procedures associated with the windows that the object creates must be able to process DM_ messages.

Target objects are not necessarily able to process every type of source object that is dropped on them. They are, however, capable of processing more than one type of dropped source object. Printer objects, for example, cannot print binary files, but they can print both text and graphics files. Because of differing capabilities, each target object should determine if it can process the source object being dropped on it. The following table shows the direct manipulation messages that are sent to target objects and the methods that are invoked by the Workplace Shell:

Description	Method Name	Invoked On	Message Name
Format drag information	wpFormatDragItem	Source	None
Request rendering format	wpRender	Source	DM_RENDER
Rendering request completed	wpRenderComplete	Target	DM_RENDERCOMPLETE
Objects being dragged over	wpDragOver	Target	DM_DRAGOVER
Object has been dropped	wpDrop	Source	DM_DROP
Drag/Drop is complete	wpEndConversation	Target	DM_ENDCONVERSATION

The "Source" and "Target" windows in the following diagram represent windows that have been subclassed, via CnrOwnerSubclassProc, by

the Workplace Shell:



Using Direct Manipulation Methods

This section shows the sequence of method flows for a typical direct manipulation operation.

Starting a Drag Operation

Folders are containers; therefore, they receive notification from the container controls when an object is added to a container. The Workplace

Shell subclasses the owner of the container control. The Workplace Shell initiates the Drag operation, calls wpFormatDragItem to fill out the data structures, and calls DrgDrag or DrgLazyDrag to perform the actual dragging. A Workplace Shell object need not invoke DrgDrag or DrgLazyDrag, since this is done automatically by the Workplace Shell when the object completes the processing of wpFormatDragItem.

When the user begins to drag an object, the source object is notified by the system, which invokes the object's wpFormatDragItem. This method is used to build a DRAGITEM structure, which is passed to a potential target object. The DRAGITEM structure contains the source object's rendering information, which is used to determine whether a Drop operation is valid at that point. Default information for the DRAGITEM structure is inserted by the default processing provided by the parent class, but an object class may override the method and include its own class-specific processing. The DRAGITEM structure is nested within a DRAGINFO structure, which is passed to any object over which the current item is being dragged. In a situation where more than one object is being dragged, a separate DRAGITEM is produced for each object and the set of structures is combined into a single DRAGINFO structure. The DRAGINFO structure, as shown in the following code fragments contains information about the Drag operation:

```
typedef struct _DRAGINFO
{
  ULONG   cbDraginfo;             /* Size of DRAGINFO               */
  USHORT  cbDragitem;             /* Size of DRAGITEM               */
  USHORT  usOperation;            /* Current Drag operation         */
  HWND    hwndSource;             /* Window handle of source        */
  SHORT   xDrop;                  /* X coordinate of drop position  */
  SHORT   yDrop;                  /* Y coordinate of drop position  */
  USHORT  cditem;                 /* Count of DRAGITEM              */
  USHORT  usReserved;             /* Reserved for future use        */
} DRAGINFO;
```

The DRAGITEM data structure, as shown in the following code fragments is used to describe the items being dragged:

```
typedef struct _DRAGITEM
{
  HWND    hwndItem;               /* Conversation partner          */
  ULONG   ulItemID;               /* Identifies item being dragged */
  HSTR    hstrType;               /* Type of item                  */
  HSTR    hstrRMF;                /* Rendering mechanism and format*/
  HSTR    hstrContainerName;      /* Name of source container      */
  HSTR    hstrSourceName;         /* Name of item at source        */
  HSTR    hstrTargetName;         /* Suggested name of item at dest*/
  SHORT   cxOffset;               /* X offset of the origin of the */
                                  /* image from the mouse hotspot  */
  SHORT   cyOffset;               /* Y offset of the origin of the */
                                  /* image from the mouse hotspot  */
  USHORT  fsControl;              /* Source item control flags     */
  USHORT  fsSupportedOps;         /* Ops supported by source       */
} DRAGITEM;
```

The Workplace Shell itself performs the intialization of the DRAGINFO structure. If class-specific processing is required, the object may perform its own initialization during processing of wpFormatDragItem. A Workplace Shell object does not need to allocate the DRAGITEM structure, since it is allocated by the Workplace Shell, and a pointer is passed to wpFormatDragItem. The following sample code shows an example of how to use wpFormatDragItem:

```
/*
 *  Specify a unique RMF that will only be understood
 *  by my instances of class MYFOLDER.
 */

SOM_Scope BOOL SOMLINK myf_wpFormatDragItem(MYFILE *somSelf,
                PDRAGITEM pdrgItem)

{
  /* MYFILEData *somThis = MYFILEGetData(somSelf); */
  MYFILEMethodDebug("MYFILE","myf_wpFormatDragItem");

  parent_wpFormatDragItem(somSelf,pdrgItem);

  /* We do NOT want to really let the workplace shell render
   * our object, so change the rendering mechanism and format
   * to be uniquely ours.
   */

  DrgDeleteStrHandle(pdrgItem->hstrRMF);
  pdrgItem->hstrRMF = DrgAddStrHandle("<DRM_OUROWNSPECIAL,DRF_OBJECT>");
  return(TRUE);
}
```

Determining whether Data Can Be Exchanged

During the drag portion of a Drag and Drop operation, the target must determine if it can exchange or receive data from the source for each object involved in the operation. The target determines which rendering mechanism and format to use. There are four combinations of rendering mechanisms and formats supported by the Workplace Shell:

- <DRM_OBJECT, DRF_OBJECT>
- <DRM_OS2FILE, DRF_UNKNOWN>
- <DRM_DISCARD, DRF_UNKNOWN>
- (DRM_OS2FILE, DRM_PRINT) X (DRF_TEXT)

The DRAGITEM structure contains the source object's rendering information, which is used to determine whether a Drop operation is valid at that point. The *pDragItem hstrRMF* field in the DRAGITEM data structure is used to hold a handle to a string containing the rendering mechanism and format.

DRM_OBJECT

The Workplace Shell interface uses the rendering mechanism of <DRM_OBJECT, DRF_OBJECT> to communicate information about the Workplace Shell objects involved in a direct manipulation.

For the <DRM_OBJECT, DRF_OBJECT> rendering mechanism, *pDragInfo hwndSource* is expected to be the window handle of the container holding objects which were inserted with wpCnrInsertObject.

When you are using the DRM_OBJECT rendering mechanism, *pDragItem ulItemID* indicates the object being dragged. For each item being dragged, *pDragItem ulItemID* is the PMINIRECORDCORE or PRECORDCORE pointer associated with the object. The object can be retrieved using OBJECT_FROM_PREC on *pDragItem ulItemID* . As soon as the target object has completed processing, a DM_ENDCONVERSATION message is sent to the window handle in *pDragItem hwndItem* .

Note: There is no way to force source rendering for DRM_OBJECT.

DRM_OS2FILE

This rendering mechanism can be used by various containers, including file folders and trash cans. These containers allow objects to be dragged and dropped on white space in the container to accomplish a **Move** or **Copy** operation. They also can allow objects in the same or another container to be dragged and dropped on objects within the container to accomplish an operation.

When you are using DRM_OS2FILE, *pDragItem ulItemID* is not used.

If the Workplace Shell is to do target rendering, **(pDragItem hstrContainerName)* is specified. It is the name of the directory that contains the file or the directory being dragged. Leave this field NULL if you want the Workplace Shell to send DM_RENDER messages, that is, to do source rendering for this object.

**(pDragItem hstrSourceName)* is the file or directory name being dragged and must be specified.

**(pDragItem hstrTarget)* is the suggested name of the file or directory at the target.

Note: If *pDragItem hstrContainerName* and *pDragItem hstrSourceName* are both specified, the Workplace Shell converts the item being dragged to an object and treats it like a DRM_OBJECT.

Messages

The following messages are used by the DRM_OS2FILE:

- DM_RENDER

 This message is sent by a target to a source, via wpRender, to request a rendering for an object. When this message is received, the source determines if it understands the rendering mechanism and format selected by the target for the object. It also confirms that it allows the operation selected by the user for that object. The source must respond to this message before proceeding with the rendering operation.

- DM_RENDERCOMPLETE

 This message is posted by a source to a target, via wpRenderComplete, to notify the target that the rendering operation has been completed by the source, either successfully or unsuccessfully. The source can elect to let the target retry a successful or an unsuccessful operation. In this case, it should return to its state at the time of the drop for that object and indicate, in the message, that a retry is allowed. Support for this message by a source is optional. If this message is not supported, then:

 - The source must convey all necessary information to the target order to allow it to handle the rendering operation.

 - It must always indicate that native rendering is allowed when replying to a DM_RENDER message.

- DM_ENDCONVERSATION

 This message is sent by a target to a source to notify the source that the rendering operation is complete and that the conversation is terminated. When this message is received, the entire Drop operation for the object is complete. The source can now release any resources it had allocated to the drop and rendering operations. When the reply is received, the target can release the resources it had allocated to the operation.

Dragging over Another Object

When an object is dragged over another object, the system invokes wpDragOver in the object being dragged over. This method receives a DRAGINFO structure, which contains pointers to DRAGITEM structures and other information. The potential target object examines this information to determine the validity of a Drop operation, and if the drop is not valid, changes the pointing device to indicate a "do not drop" situation. The following sample code shows an example of how to use wpDragOver:

```
/*
 *  Rejects objects that are not file system objects from being dropped
 *  on the Browse_O_matic.
 */

SOM_Scope MRESULT SOMLINK Browse_O_Maticwps_wpDragOver(
                          Browse_O_Matic *somSelf,
                          HWND hwndCnr,
                          PDRAGINFO pdrgInfo)
{
  MRESULT mResult;
  ULONG   ulCount;
  ULONG   ulNumberOfObjects;

  /* Browse_O_MaticData *somThis = Browse_O_MaticGetData(somSelf); */
  Browse_O_MaticMethodDebug("Browse_O_Matic",
                            "Browse_O_Maticwps_wpDragOver");

  /* Don't call the parent. Initialize mResult to allow */
  /* the drag over to proceed                           */
  mResult = MRFROM2SHORT(DOR_DROP, DO_MOVE);

  /* Determine the number of objects dragged */
  /* over the Browse-O-Matic                 */
  ulNumberOfObjects = DrgQueryDragitemCount( pdrgInfo);

  /* Check all the objects */
  for (ulCount=0; ulCount < ulNumberOfObjects &&
                            SHORT1FROMMR(mResult) !=
                                        DOR_NEVERDROP; ulCount++)
```

```
  {
    /* It must be a file system type object */
    if(DrgVerifyRMF(DrgQueryDragitemPtr(pdrgInfo, ulCount),
                    "DRM_OS2FILE",
                     NULL))
      mResult = MRFROM2SHORT(DOR_DROP, SHORT2FROMMR(mResult));
    else
      mResult = MRFROM2SHORT(DOR_NEVERDROP, SHORT2FROMMR(mResult));
  }
  return(mResult);
}
```

Ending a Drop Operation

When a Drop operation occurs, the object being dropped upon is notified by the system, which invokes wpDrop. This method accepts the DRAGINFO structure and determines whether the information is sufficient to take action, or whether further conversation is necessary to determine rendering information.

When Dropping on a Folder

When dropping DRM_OBJECT or DRM_OS2FILE on a folder, with *pDragItem hstrContainerName* or *pDragItem hstrSourceName* defined, the Workplace Shell will perform the action, for example, **Copy**, **Move**, **Shadow**, and notify the source with a DM_ENDCONVERSATION message.

When dropping DRM_OS2FILE on a folder, with *pDragItem hstrContainerName* set to NULL, the Workplace Shell will initiate source rendering by sending a DM_RENDERPREPARE message, if DC_PREPARE is specified in the *pDragItem fsControl* word in the DRAGITEM structure. It then sends a DM_RENDER message. Again, DM_ENDCONVERSATION is sent as previously described.

When Dropping on a Printer

Workplace Shell implements the standard PM Print rendering mechanism, DRM_PRINT. When dropping DRM_OBJECT or DRM_OS2FILE on a printer, with *pDragItem hstrContainerName* or *pDragItem hstrSourceName* defined, the Workplace Shell will perform the action, and notify the source with a DM_ENDCONVERSATION message.

When dropping DRM_OS2FILE, with *pDragItem hstrContainerName* set to NULL, or dropping DRM_PRINT on a printer, the printer sends a DM_PRINTOBJECT message.

When Dropping on the Shredder

The Workplace Shell shredder object supports a rendering mechanism called DRM_DISCARD, not a standard PM mechanism. When dropping DRM_OBJECT or DRM_OS2FILE on a shredder, with *pDragItem hstrContainerName* or *pDragItem hstrSourceName* defined, the Workplace Shell will perform the action, and notify the source with a DM_ENDCONVERSATION message.

When dropping DRM_OS2FILE, with *pDragItem hstrContainerName* set to NULL, or dropping DRM_DISCARD on the shredder, the shredder sends a DM_DISCARDOBJECT message.

The following sample code shows an example of how to use wpDrop:

```
/*
 *  Rejects objects that are not file system objects from being dropped
 *  from being dropped on the Browse_O_matic.
 */
```

```
SOM_Scope MRESULT SOMLINK Browse_O_Maticwps_wpDrop(
                          Browse_O_Matic *somSelf,
                          HWND hwndCnr,
                          PDRAGINFO pdrgInfo,
                          PDRAGITEM pdrgItem)

{
  CHAR    pszBuffer[255];
  MRESULT mResult;

  /* Browse_O_MaticData *somThis = Browse_O_MaticGetData(somSelf); */
  Browse_O_MaticMethodDebug("Browse_O_Matic",
                            "Browse_O_Maticwps_wpDrop");

  mResult = MRFROM2SHORT(DOR_DROP, 0);

  /* Don't call the parent. Initialize mResult to allow */
  /* the drop to proceed.                               */
  if(DOR_NEVERDROP != SHORT1FROMMR(mResult) &&
                      DrgVerifyRMF(pdrgItem, "DRM_OS2FILE", NULL))

  {
    /* Get the path */
    DrgQueryStrName(pdrgItem->hstrContainerName,
                    sizeof(pszBuffer),
                    pszBuffer);

    /* Append the name of the object to the path */
    DrgQueryStrName(pdrgItem->hstrSourceName,
                    sizeof(pszBuffer) - strlen(pszBuffer),
                    &pszBuffer[strlen(pszBuffer)]);
    _wpViewObject(somSelf, NULLHANDLE, OPEN_DEFAULT, (ULONG)pszBuffer);
    mResult = MRFROM2SHORT(DOR_DROP, SHORT2FROMMR(mResult));
  }
  else
    mResult = MRFROMSHORT(DOR_NEVERDROP);

  return(mResult);
}
```

Object Persistence: Save/Restore Methods

The WPObject class provides methods that support persistent objects, that is, objects for which the system saves information. These methods include save and corresponding restore methods. This chapter provides information for using these state methods in Workplace Shell applications.

About Save/Restore Methods

A Workplace Shell object can exist in two states:

- Dormant: persistent form of object when it is stored away on disk. If the object's class is a descendant of WPFileSystem, its data is stored in the object's extended attributes. If it is a descendant of WPAbstract, its data is stored in the .INI file.
- Awakened: object is instantiated as a SOM object that can have methods invoked upon it.

The wpSaveState method will save the value of certain instance variables to a persistent storage location. The wpRestoreState method will restore these values for a newly awakened object. The Workplace Shell uses these methods to maintain important instance data for each object. If you write a subclass of a Workplace Shell class that includes its own new instance data, you must override these methods to ensure that the new instance data values are saved/restored along with the rest of the object's data.

The following events trigger the Save/Restore methods:

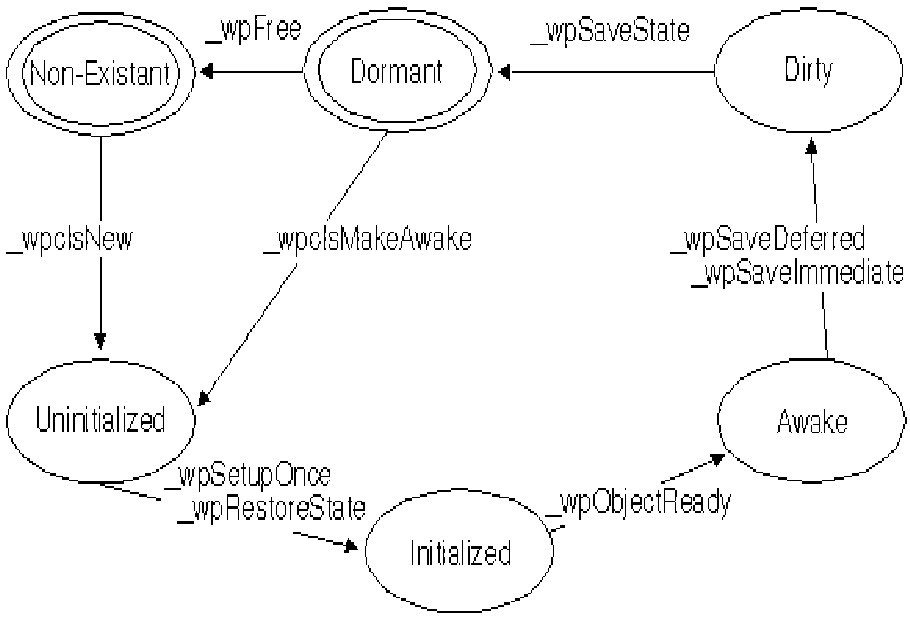
- Workplace Shell calls wpRestoreState when an object is awakened.
- The wpSaveImmediate method calls wpSaveState to save its data to disk.

- An application must save an object to disk when a critical instance variable is changed. The wpSaveImmediate method performs the save on the Workplace User-interface thread, and is therefore a synchronous operation. The wpSaveDeferred method causes the save to take place on a separate thread, asynchronously. Therefore, for performance reasons, wpSaveDeferred should be used in preference to wpSaveImmediate.

- Workplace Shell calls wpSaveDeferred to asynchronously save the object's state when:

 - An object is closed
 - An object is made dormant
 - The system is shut down.

The following figure shows the life cycle of an object:



If you are implementing a subclass of a Workplace Shell class with its own instance variables that are supposed to persist, override wpSaveState and wpRestoreState. In the override for wpSaveState, call wpSaveData, wpSaveLong, or wpSaveString, for each piece of data you want saved, depending on the type of each piece of instance data associated with the object. Likewise, the override for wpRestoreState should call wpRestoreData, wpRestoreLong, or wpRestoreString, for each piece of data to be restored.

The wpSaveState and wpRestoreState methods use a key to identify each piece of data for an object. This key is made up of the class name and a unique ULONG. Together, this key should uniquely identify a given piece of data for the object. For user-defined subclass instance data, assign a ULONG to each piece of data. Supply that ULONG along with your class name to wpSaveData, wpSaveLong, or wpSaveString, when saving that piece of data. Likewise, supply that ULONG along with your class name to wpRestoreData, wpRestoreLong, or wpRestoreString, when restoring that piece of data.

Using Save/Restore Methods

The following sample codes (IDL and C files) serves as an example of how to use the Save/Restore methods. In this example, class X, a subclass of WPDataFile, introduces two pieces of persistent instance data.

```
include wpdataf.idl

interface X : WPDataFile
{
  .
  .
  .

  implementation
  {

    ULONG ulA;     //# Persistent instance data introduced by class X
    PSZ   pszB;

    wpSaveState: override;    //# To save your data
```

```
    wpRestoreState: override; //# To restore your data
    wpInitData: override;     //# To supply default values for your data
  };
};
```

The following sample code shows the source code for X:

```
BOOL SOMLINK x_wpSaveState(X * somSelf)
{
  _wpSaveLong(somSelf, "X", 1, _ulA);
  _wpSaveString(somSelf, "X", 2, _pszB);   // Key is X and 1 for _ulA
  return parent_wpSaveState(somSelf);      // Key is X and 2 for _pszB
}
  .
  .
  .

BOOL SOMLINK x_wpRestoreState(X * somSelf, ULONG ulReserved)
{
  ULONG ulSize;

  // Key is X and 1 for _ulA
  _wpRestoreLong(somSelf, "X", 1, &_ulA);

  // Key is X and 2 for _pszB
  _wpRestoreString(somSelf, "X", 2, &_pszB, &ulSize);
  return parent_wpRestoreState(somSelf, ulReserved);
}
  .
  .
  .

BOOL SOMLINK x_wpInitData(X * somSelf)
{
  BOOL rc;
  rc    = parent_wpInitData(somSelf);
  _ulA  = 1001;
  _pszB = "hello";
  return rc;
}
```

Note: Some of the save and restore methods are used in a complete program that is illustrated in Sample Code for Setup/Cleanup Methods.

Object Properties: Notebook Methods

A notebook is a visual component used to display the setttings for an object and to enable the user to change them. This chapter describes the General and Window pages of the Settings notebook, and explains how to use the notebook methods in Workplace Shell applications.

About Notebook Methods

WPObject defines a General page in the Settings notebook for all classes of objects. Because all Workplace Shell objects have general properties associated with them, therefore, all Workplace Shell objects have a General page in their Settings notebook. The General page describes all the general object properties such as the object's title, icon, and whether or not the object is a template.

The title or icon of an object can be changed using the General page. For WPAbstract objects, the title and icon change is saved in the OS2.INI file. For the WPFileSystem objects, the title change is saved in the .LONGNAME extended attributes, and the icon change is saved in the .ICON extended attributes.

WPObject also defines a Window page in the Settings notebook for all classes of objects. Most of the Abstract objects have both General and Window pages in the Settings notebook. Some Abstract objects like mouse and keyboard do not have a Window page because they have no open views other than Settings.

Settings notebook pages for Workplace Shell objects are inherited from the ancestors of the Workplace Shell class. This means that they include pages that have been added or removed by their ancestor classes, in addition to the General and Window pages inherited from the root, WPObject class.

For example, suppose that MyObject is a persistent object derived from the WPAbstract class. Because WPAbstract is derived from the WPObject class, MyObject inherits the characteristics and behaviors of WPAbstract and WPObject. WPAbstract inherits its Settings notebook from WPObject. MyObject, therefore, inherits WPObject's Settings notebook.

Now suppose that MyObject class defines two additional pages in its Settings notebook, MyPage_1 and MyPage_2. This means that the Settings notebook for MyObject class has four pages:

- General
- Window
- MyPage_1
- MyPage_2

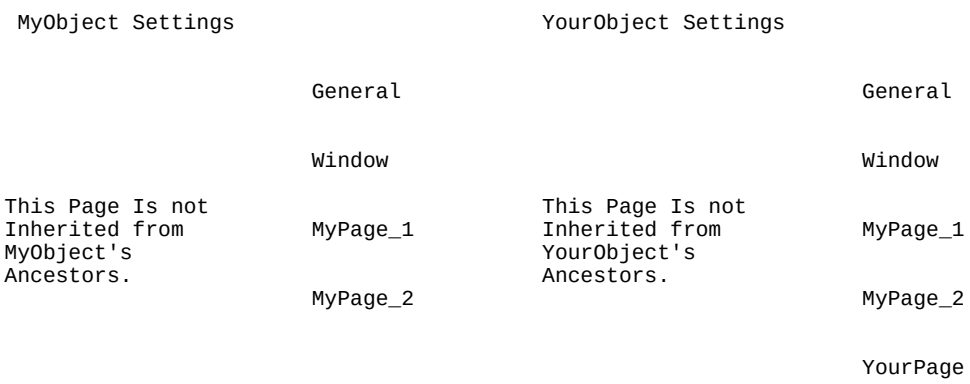
Consider that YourObject class is derived from MyObject class. Therefore, by inheritance, YourObject class defines the following pages:

- General
- Window
- MyPage_1
- MyPage_2

in its Settings notebook. YourObject class defines an additional page in its Settings notebook, YourPage. The Settings notebook for YourObject class has now five pages:

General	Inherited from the WPObject class.
Window	Inherited from the WPObject class.
MyPage_1	Inherited from MyObject class.
MyPage_2	Inherited from MyObject class.
YourPage	Defined by YourObject.

The tabs in the Settings notebook for MyObject and YourObject are shown in the following figure:



Every Settings notebook in the Workplace Shell should have Undo, Default, and Help push buttons. Any changes to the Settings notebook takes effect immediately, although sometimes the changes are delayed until the focus is changed.

Using Notebook Methods

This section describes how the Settings notebook methods are used to add or to remove pages that a class has inherited from its ancestors' Settings notebook.

Inserting Notebook Pages

An object's Settings notebook is created by the Workplace Shell by calling the object's wpAddSettingsPages. Adding pages to the Settings notebook that a class has inherited from its ancestors is accomplished by overriding wpAddSettingsPages and by calling a new method that inserts the page. The new method calls wpInsertSettingsPage to insert the page into the object's notebook. The following sample code shows how to insert pages to an object's notebook:

```
/*************************  New Method   ***************************/

SOM_Scope ULONG SOMLINK MyObject_wpAddAnotherPage(
                        MyObject *somSelf, HWND hwndNotebook)

{
  PAGEINFO pageinfo;
  .
  .
  /* Setup the page information data structure for my new page */
  .
  .
  /* Add the page to the Settings notebook */
  return _wpInsertSettingsPage(somSelf, hwndNotebook, &pageinfo);
}

/************************  Method Overrides  ************************/

SOM_Scope BOOL32 SOMLINK MyObject_wpAddSettingsPage (
                         MyObject *somSelf, HWND hwndNotebook)

{
  .
  .

  if (parent_wpAddSettingsPage (somSelf, hwndNotebook)
      && _wpAddAnotherPage (somSelf, hwndNotebook))
    return (TRUE);   /* Page added successfully */
  else
    return (FALSE);  /* Something failed */
}
```

New pages for an object's Settings notebook can be placed at the top or bottom of pages inherited from the object's ancestor classes. By calling parent_wpAddSettingsPages before calling the new wpAddAnotherPage, the new page is added to the top of the Settings notebook, above pages inherited from ancestor classes. If the sequence is reversed, the new page is added to the bottom of the Settings notebook, below pages inherited from ancestor classes.

When a new page is added to the Settings notebook, any changes made to the object's instance data can be refreshed from the file system, via wpCnrRefreshDetails. This instance method causes all currently visible RECORDCORE structures to be refreshed with the current object details.

Removing Notebook Pages

A page can be removed from an object's Settings notebook by overriding the ancestors' method that inserts it. The Settings notebook for YourObject class inherits the General, Window, MyPage_1, and MyPage_2 pages defined by its ancestors, MyObject and WPObject. YourObject class might have requirements for MyPage_1 but not MyPage_2. To remove MyPage_2 from YourObject's Settings notebook, YourObject must override the method inherited from MyObject that adds MyPage_2 to the Settings notebook, and return SETTINGS_PAGE_REMOVED without calling the parent method. The following sample code shows how to remove a page from an object's notebook:

```
/*********************  Method Overrides  ***************************/

SOM_Scope ULONG SOMLINK YourObject_wpAddAnotherPage(
                           YourObject *somSelf HWND hwndNotebook)
{
  .
  .
  .

  /* Remove the page from the Settings notebook */
```

```
  return (SETTINGS_PAGE_REMOVED);
}
```

The same technique can be used to replace or to remove the General page from an object's Settings notebook by overriding wpAddSettingsPages and wpAddObjectGeneralPage. The override to wpAddSettingsPages calls wpAddObjectGeneralPage. To remove the General page, the override to wpAddObjectGeneralPage returns SETTINGS_PAGE_REMOVED without calling the parent method. To replace the General page with another page, the override to wpAddObjectGeneralPage calls wpInsertSettingsPage without calling the parent method.

Object Usage: Usage Methods

Every Workplace Shell object in the system has an in-use list. Object usage methods allow an object to keep track of its resources and how it is being used. This chapter provides information for using in-use lists that are associated with objects.

About Usage Methods

The in-use list provides the object with information such as:

- The number of container windows into which the object has been inserted (USAGE_RECORD)
- The number of open views (contents and settings) of the object that already exist (USAGE_OPENVIEW)
- How much memory the object has allocated (USAGE_MEMORY)
- How many awakened shadows there are of the object (USAGE_LINK)
- Which application opened the object (USAGE_OPENFILE). Note that this information only applies to data file objects.

All objects that the user interacts with are actually just records that have been inserted into a container control. A container is an object, such as a folder, which also has an in-use list. Because of the close relationship between an object's in-use list and container objects, this chapter includes a description of methods that are relevant to container usage as well as object usage. Object Usage methods, as shown in the following table, allow an object to keep track of its resources and how it is being used:

Method Name	Description
wpAddToObjUseList	Adds an object to the list of objects in use by the object.
wpAssertObjectMutexSem	This instance method verifies that an object's mutex semaphore is held for the current thread.
wpCnrDeleteUseItem	Deletes the object's USAGE_RECORD use-item for the specified container.
wpDeleteFromObjUseList	Deletes an object from the list of objects in use by the object.
wpFindUseItem	Retrieves an object from the list of objects in use by the object.
wpFindViewItem	Retrieves a USAGE_OPENVIEW item from the list of items in use by the object.
wpIsLocked	Tells you whether the object is locked or not.
wpLockObject	Increases the lock counter for the object. Objects that are locked will not be made dormant.
wpReleaseObjectMutexSem	Releases the mutex semaphore for an object. Each object has associated with it a mutex semaphore that can

	be used to serialize access to resources.
wpRequestObjectMutexSem	Requests the mutex semaphore for an object. Each object has associated with it a mutex semaphore that can be used to serialize access to resources.
wpUnlockObject	Unlocks the object so that it can be made dormant.

The in-use list is a linked list of USEITEM data structures. It consists of an item type and a pointer to the next USEITEM data structure, immediately followed by an item type-specific data structure. The types of items that can be added to an object's in-use list and the type-specific data structures that follow each USEITEM data structure are shown in the following table:

Item	Structure	Description
USAGE_LINK	LINKITEM	A shadow of the object has been awakened.
USAGE_MEMORY	MEMORYITEM	Memory has been allocated by wpAllocMem.
USAGE_OPENFILE	VIEWFILE	The object (if a data file) has been opened.
USAGE_OPENVIEW	VIEWITEM	A view of the object has been opened.
USAGE_RECORD	RECORDITEM	The object has been inserted into a container window.

Using Usage Methods

This section describes the different usage methods along with explaining how each of them relate to an in-use item.

Adding Items to the Object's In-Use List

The wpAddToObjUseList method adds items to the object's in-use list. When an object's memory is allocated, wpAllocMem calls wpAddToObjUseList to add a USAGE_MEMORY item to the object's in-use list. The allocated memory immediately follows the MEMORYITEM data structure. The following figure shows the syntax of the MEMORYITEM data structure:

```
typedef struct _MEMORYITEM
{
  ULONG   cbBuffer;   /* Number of bytes in memory block */
}MEMORYITEM;
```

The wpOpen method calls wpAddToObjUseList to add a USAGE_OPENVIEW item to the object's in-use list. The wpSwitchTo method scans the in-use list for USAGE_OPENVIEW items to give focus to an already existing view. In general, wpViewObject is used instead of wpOpen because wpViewObject considers the setting in the Object open behavior group-box control on the Window page of the Settings notebook, and calls either wpOpen or wpSwitchTo. The following figure shows the syntax of the VIEWITEM data structure:

```
typedef struct _VIEWITEM
{
  ULONG   view;              /* Object view that this represents */
  LHANDLE handle;            /* Open handle                      */
```

```
  ULONG ulViewState;         /* View state flags                */
  HWND hwndCnr;              /* Used only by system             */
  PMINIRECORDCORE pRecord;   /* Used only by system             */
}VIEWITEM;
```

When wpSetLinkToObject is called to link a shadow to the object it points to, wpAddToObjUseList is called to add the USAGE_LINK item to the original object's in-use list. The following figure shows the syntax of the LINKITEM data structure:

```
typedef struct _LINKITEM
{
  WPObject *LinkObj;         /* The link object  */
}LINKITEM;
```

When a data file is opened, its associated program is started and a USAGE_OPENFILE use-item is added to the data file's in-use list. The following figure shows the syntax of the VIEWFILE data structure:

```
typedef struct _VIEWFILE
{
  ULONG   ulMenuId;          /* Menu id if association or menu page */
  LHANDLE handle;            /* Open handle                         */
  HWND    hwndCnr;           /* Used only by system                 */
  PMINIRECORDCORE pRecord;   /* Used only by system                 */
}VIEWFILE;
```

Removing Items from the Object's In-Use List

When wpCnrInsertObject puts an object into a container window, its adds a USAGE_RECORD item to the object's in-use list. Conversely, when wpCnrRemoveObject is called, the USAGE_RECORD item is deleted from the object's in-use list and the object is removed from the container. The following figure shows the syntax of the RECORDITEM data structure:

```
typedef struct _RECORDITEM
{
  HWND   hwndCnr;            /* Container into which object      */
                             /* is being inserted                */
  PMINIRECORDCORE pRecord;   /* Record pointer within container  */
  ULONG  ulUser;             /* For application use              */
}RECORDITEM;
```

Likewise, when wpFreeMem is called to free memory, it calls wpDeleteFromObjUseList to delete a USAGE_MEMORY item from the object's in-use list.

When views are closed by the object's wpClose, wpDeleteFromObjUseList removes the applicable USAGE_OPENVIEW item from the object's in-use list. The wpCnrDeleteUseItem method deletes the USAGE_RECORD use-item for the specified container, but does not remove the object from the container. Use wpCnrRemoveObject to delete the USAGE_RECORD use-item and also to remove the object from the container.

Increasing or Decreasing the Object's Lock Count

The wpLockObject method increases the object's lock count by one. The wpUnlockObject method decreases the object's lock count by one. If the lock count is zero, the object can be made dormant. The wpIsLocked method returns TRUE if the object is locked. Otherwise, it returns FALSE.

Finding an Object's In-Use Item

You can use wpFindUseItem to determine how an object is being used. It searches an object's in-use list for items of a specified type and returns a pointer to the USEITEM data structure that matches that specified type. The wpFindViewItem method finds USAGE_OPENVIEW items in the use list. The following figure shows the syntax of the USEITEM data structure:

```
typedef struct _USEITEM
{
  ULONG   type;                 /* Type of this item     */
  struct _USEITEM FAR *pNext;   /* Next item in use list */
}USEITEM;
```

The following sample code shows an example of how you can use the usage methods. It uses the wpFindUseItem and wpFindViewItem along with the USAGE_OPENVIEW item.

```
SOMAny    *Object;
PVIEWITEM pViewItem;
PUSEITEM  pUseFile;
ULONG     ulView;
// Are there any open views of this object?
if (_wpFindUseItem(Object, USAGE_OPENVIEW, NULL))
{
  // Find any views of this object
  for (pViewItem = _wpFindViewItem(Object, VIEW_ANY, NULL);
       pViewItem;
       pViewItem = _wpFindViewItem(Object, VIEW_ANY, pViewItem))
  {
    // Is a program running?
    if (pViewItem->view == OPEN_RUNNING)
    {
      ulView = (ULONG)0;

      // Find any open file use-items
      for (pUseFile = _wpFindUseItem(Object, USAGE_OPENFILE, NULL);
           pUseFile;
           pUseFile = _wpFindUseItem(Object, USAGE_OPENFILE, pUseFile))
      {
        // Does the VIEWFILE's open handle match the VIEWITEM's open handle?
        if (((PVIEWFILE)(pUseFile+1))->handle == pViewItem->handle)
        {
          // Save the VIEWFILE's menu id
          ulView = ((PVIEWFILE)(pUseFile+1))->ulMenuId;
          break;
        }
      }
    }
  }
}
```

Object User Action: Pop-Up Menu Methods

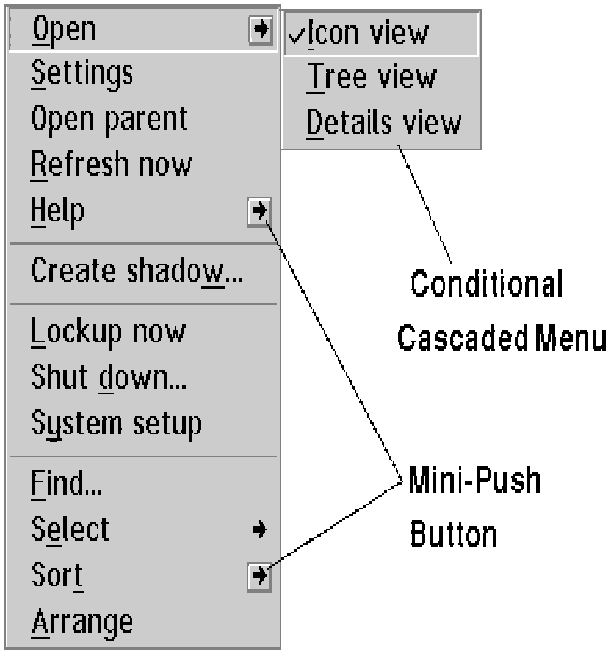
This chapter describes the pop-up context menus that can be used to invoke actions on Workplace Shell objects. It also provides an overview of how to create and manipulate the pop-up context menu for your Workplace Shell objects.

About Pop-Up Menu Methods

Pop-up menu methods support the actions that the user can perform on an object. These actions appear in a context or pop-up menu when the user presses button 2 of the pointing device. A pop-up menu contains action choices for an object in its current context or state. The contents of a pop-up menu depends on the state of the object.

Pop-up menus consist of a set of selectable items and any pull-down or conditional cascaded menus associated with them. In the following figure, **Open**, **Settings**, **Open parent**, **Refresh now**, and so on, are items in the object's primary pop-up menu. **Icon**, **Tree**, and **Details** are

items in the **Open** pull-down or conditional cascaded menu.



Conditional cascaded menus have mini-push buttons displayed next to the pop-up menu item. When the user selects the mini-push button, the pull-down menu is displayed. As shown in the above figure, **Open**, **Help**, and **Sort** have mini-push buttons that are represented by a right arrow () surrounded by a square box. If the user selects one of these three pop-up menu items, a default action listed in the submenu is performed. The default action is represented by a check mark. The default action for the **Open** item is Icon view, the **Help** item is General help, and the **Sort** item is Name. This submenu is called a *conditional cascaded* menu because it is displayed only if the user selects the mini-push button.

Like Settings notebook pages, pop-up menus are inherited from a class' ancestor classes. This means that they include pop-up menu items that ancestor classes have added to or removed from the pop-up menu inherited from WPObject. The pop-up methods permit you to add new menu items to or remove menu items from the pop-up menu inherited from an object's ancestor classes, as shown in the following table:

Method	Description
wpAddUserItemsToPopupMenu	Adds user-defined items to the popup menu.
wpFilterMenu	Filters out options from the object's pop-up menu that do not apply. (OS/2 Version 4 or higher.)
wpFilterPopupMenu	Filters out options from the object's pop-up menu that do not apply.
wpInsertMenuItems	Inserts items into the object's pop-up menu. (OS/2 Version 4 or higher.)
wpInsertPopupMenuItems	Inserts items into the object's pop-up menu.
wpModifyMenu	Adds new options to the object's pop-up menu. (OS/2 Version 4 or higher.)
wpModifyPopupMenu	Adds new options to the object's pop-up menu.
wpSetMenuStyle	Sets the menu style to be either long or short.

When the user requests an object's pop-up menu, the Workplace Shell builds it by calling the object's wpFilterPopupMenu and wpModifyPopupMenu. The wpInsertPopupMenuItems method is called by an override to wpModifyPopupMenu to add new options to an object's pop-up menu.

Support for User Selection of Standard Pop-Up Menu Items

When the user selects a standard action from a pop-up menu, the Workplace Shell calls one of the pop-up menu methods shown in the following table:

Method Name	Description
wpAppendObject	Appends a file system object to another file system object.
wpClose	Closes all open views of an object.
wpCopyObject	Creates a new copy of the object.
wpCreateFromTemplate	Creates an object from a template.
wpCreateShadowObject	Creates a shadow of an object.
wpDelete	Deletes an object and prompts for confirmation, if necessary.
wpDisplayHelp	Displays a help panel.
wpHide	Hides or minimizes open views of an object.
wpMoveJobObject	Moves a print job to a different printer.
wpMoveObject	Moves the object to a different location.
wpOpen	Opens a view of the object.
wpPrintObject	Prints a view of the object.
wpReplaceObject	Replaces an object in another folder.
wpRestore	Restores hidden or minimized views of an object.

Open Views

Objects can have open actions or open views associated with them. *Open views* typically are the views of an object, for example, Icon, Tree, Details, and Settings. Open views, for data file objects, also include programs or program references that the user has associated with the object. Open views are displayed when the user selects the cascaded mini-push button that appears next to the **Open** action on the pop-up menu. The user then can select the default open view or any of the open views listed in the conditional cascaded menu. The Workplace Shell defines a set of predefined open views, as shown in the following table:

View	Description
OPEN_CONTENTS	Opens content view.
OPEN_DEFAULT	Opens default view.
OPEN_DETAILS	Opens Details view.
OPEN_HELP	Opens help view.
OPEN_RUNNING	Executes object.
OPEN_SETTINGS	Opens Settings notebook.
OPEN_TREE	Opens Tree view.
OPEN_USER	Opens class-specific view (value greater than OPEN_USER)

Some of the predefined open views are meaningful only to certain Workplace Shell classes:

- OPEN_RUNNING is meaningful only to a program or program reference object
- OPEN_TREE is meaningful only to file system objects such as folders, drives and directories.

Workplace Shell classes can define new open views for their objects by:

1. Adding the **New view** menu item to the **Open** submenu.
2. Overriding wpMenuItemSelected to support user selection of the **New view** menu item.
3. Overriding wpOpen to open the **New view**.
4. Creating and opening a standard window for the **New view** by calling WinCreateStdWindow.

 Note: The preferred method for displaying application views of an object is for the object to start a separate process, using DosExecPgm, for the application. This approach moves the larger part of the application code out of the Workplace Shell's process, thus conserving the its resources. It also helps prevent a misbehaved application from potentially interfering with the execution of the Workplace Shell.
5. Adding a USAGE_VIEW item to the object's in-use list by calling wpAddToObjUseList.
6. Registering the **New view** by calling wpRegisterView.

Note: The Sample Code for Pop-Up Menu Methods is a complete program that demonstrates how to define and open a new open view.

Using Pop-Up Menu Methods

This section describes the following:

- Adding and removing standard pop-up menu items
- Adding and removing items to a pop-up menu inherited from an object's ancestors
- Adding conditional cascaded menus to a pop-up menu
- Supporing user selection of new pop-up menu items.

Note: A complete program is provided to demonstrate how to customize a Workplace Shell pop-up menu. The program is illustrated in Sample Code for Pop-Up Menu Methods.

Adding and Removing Standard Pop-Up Menu Items

The pop-up menu of a Workplace Shell object consists of a subset of the standard pop-up menu items and any new menu items defined for the object's class or inherited from other ancestors. The WPObject class defines a set of standard pop-up menu items that are inherited by all Workplace Shell objects. The WPDesktop, WPFolder, WPPalette, and WPProgram Workplace Shell classes define standard pop-up menu items for their descendants. Each standard pop-up menu item is associated with a flag, as shown in the following table:

```
Class               Item Flag           Description

WPDesktop           CTXT_LOCKUP         Open Lockup now

CTXT_SHUTDOWN       Open Shut down

WPFolder            CTXT_ARRANGE        Open Arrange
```

```
CTXT_DETAILS         Open Details view
CTXT_FIND            Open Find dialog
CTXT_ICON            Open Icon view
CTXT_SELECT          Open Select
CTXT_SORT            Open Sort dialog
CTXT_TREE            Open Tree view
WPObject             CTXT_ARRANGE         Open Arrange
CTXT_CLOSE           Close
CTXT_COPY            Copy
CTXT_CRANOTHER       Create another
CTXT_DELETE          Delete
CTXT_DETAILS         Open Details view
CTXT_HELP            Help
CTXT_ICON            Open Icon view
CTXT_LINK            Create shadow
CTXT_LOCKUP          Open Lockup dialog
CTXT_MOVE            Move
CTXT_NEW             Create another
CTXT_OPEN            Open
CTXT_PALETTE         Open Palette
CTXT_PICKUP          Pick up an object
CTXT_PRINT           Print
CTXT_PROGRAM         Open program
CTXT_PUTDOWN         Put an object down
CTXT_REFRESH         Refresh
CTXT_SELECT          Open Select
CTXT_SETTINGS        Open Settings
                     notebook
CTXT_SHADOW          Create shadow
CTXT_SHUTDOWN        Open Shut down
CTXT_SORT            Open Sort dialog
CTXT_SWITCHTO        Switch to
CTXT_TREE            Open Tree view
CTXT_WINDOW          Window
WPPalette            CTXT_PALETTE         Open palette
WPProgram            CTXT_PROGRAM         Open program
```

Workplace Shell classes can add or delete standard pop-up menu items from their pop-up menu by overriding wpFilterPopupMenu. The wpFilterPopupMenu method returns the flags that represent the standard pop-up menu items for the object. By removing a standard menu item from the pop-up menu, the override to wpFilterPopupMenu masks the flag that corresponds to the item being removed from the flags that represent the standard pop-up menu items inherited from the object's parent. For example, suppose that printing MyObject has no meaning. To remove the **Print** option from MyObject's pop-up menu, wpFilterPopupMenu is overridden as shown in the following figure:

```
/************************ Method Overrides ***************************/
```

```
/* Filters out any options from the pop-up menu that don't apply */
SOM_Scope ULONG SOMLINK MyObject_wpFilterPopupMenu
   (MyObject *somSelf,ULONG ulFlags, HWND hwndCnr, BOOL32 fMultiSelect)

{
  MyObjectData *somThis = MyObjectGetData(somSelf);
  MyObjectMethodDebug("MyObject","MyObject_wpFilterPopupMenu");

  /* Don't allow anyone to print MyObject  */
  return(parent_wpFilterPopupMenu(somSelf,ulFlags,hwndCnr,fMultiSelect)
                                  & ~CTXT_PRINT);
}
```

The flags that represent the standard pop-up menu items of MyObject's parent class are returned from the call to parent_wpFilterPopupMenu. To remove the **Print** option from MyObject's pop-up menu, these flags are joined to the complement of CTXT_PRINT using the AND logical operand. Conversely, if the pop-up menu of MyObject's parent class did not include the **Print** option, the **Print** option can be added to MyObject's pop-up menu by joining these flags to CTXT_PRINT using the OR logical operand.

Note: An object's pop-up menu is inherited from its ancestors. To ensure that calls to wpFilterPopupMenu belonging to the object's ancestors do not add the menu item after it is deleted, or remove the menu item after it is added, the parent_wpFilterPopupMenu is called first.

Adding and Removing Class Items to a Pop-Up Menu

New items are added to the pop-up menu inherited from an object's ancestors by overriding the object's wpModifyPopupMenu and calling wpInsertPopupMenuItems. For example, to add **New item** to MyObject's pop-up menu, the new menu item is defined in a resource file in the same manner as menus are defined in PM programs. An ID is assigned to the new menu and to the menu item, as shown in the following sample code:

```
#define ID_MOREITEMS  WPMENUID_USER+1
#define ID_NEWITEMS   WPMENUID_USER+2

MENU ID_MOREITEMS LOADONCALL MOVEABLE DISCARDABLE
  BEGIN
    MENUITEM "New Item", ID_NEWITEMS
  END
```

Menu items can also be deleted by overriding wpModifyPopupMenu and by sending the MM_DELETEITEM message to the menu.

User defined menu items must be greater than 0x6500. IDs for class-specific menus and menu items have a value greater than WPMENUID_USER, so they do not conflict with IDs for menus and menu items defined by the Workplace Shell classes, as shown in the following sample code:

```
/* Add a new item to MyObject's pop-up menu */
SOM_Scope BOOL32 SOMLINK MyObject_wpModifyPopupMenu
  (MyObject *somSelf,HWND hwndMenu, HWND hwndCnr, ULONG ulPosition)

/* Remove an item from MyObject's pop-up menu */
WinSendMsg(hwndMenu,
           MM_DELETEITEM,
           MPFROM2SHORT(MY_ITEMID,FALSE),
           NULL);

{
  /* Insert new items in MyObject's primary menu */
  _wpInsertPopupMenuItemsA(somSelf,
                           hwndMenu,
                           ulPosition,
                           hmod,
                           ID_MOREITEMS,
                           WPMENUID_PRIMARY);

  /* Add the items inherited from MyObject's parent */
  return (parent_wpModifyPopupMenu(somSelf,
                                   hwndMenu,
                                   hwndCnr,
                                   ulPosition));
```

```
}
```

The wpInsertPopupMenuItems method requires:

- A handle to the module where the menu resource is defined
- The ID for the menu resource
- The ID for the menu where the item is being inserted.

In the above sample code, ID_MOREITEMS is the ID for the menu resource that defines the new menu item being added to the object's primary pop-up menu. WPMENUID_PRIMARY is the ID for the object's primary pop-up menu, where **New item** is being inserted.

An item can be added to a pop-up submenu or conditional cascaded menu by specifying the ID for the conditional cascaded menu on the call to wpInsertPopupMenuItems. For example, to add **New item** to the **Open** conditional cascaded menu, the call to wpInsertPopupMenuItems is modified as shown in the following sample code:

```
/* Insert new items in MyObject's Open submenu */
_wpInsertPopupMenuItemsA(somSelf,
                         hwndMenu,
                         ulPosition,
                         hmod,
                         ID_MOREITEMS,
                         WPMENUID_OPEN);
```

Adding Conditional Cascaded Menus to a Pop-Up Menu

Items on an object's pop-up menu sometimes have pull-down menus or submenus associated with them. In the previous sample codes, **New item** is not a pull-down menu. However, **New item** can be defined as a pull-down menu by defining it as a submenu in MyObject's resource file, as shown in the following sample code:

```
#define ID_MOREITEMS  WPMENUID_USER+1
#define ID_NEWITEMS   WPMENUID_USER+2
#define ID_SUBITEM1   WPMENUID_USER+3
#define ID_SUBITEM2   WPMENUID_USER+4
#define ID_SUBITEM3   WPMENUID_USER+5

MENU ID_MOREITEMS LOADONCALL MOVEABLE DISCARDABLE
BEGIN
  SUBMENU "New Item", ID_NEWITEMS
  BEGIN
    MENUITEM  "SubItem_1"  ID_SUBITEM1
    MENUITEM  "SubItem_2"  ID_SUBITEM2
    MENUITEM  "SubItem_3"  ID_SUBITEM3
  END
END
```

The **New item** submenu is added to MyObject's primary pop-up menu using the same technique as shown in the "Adding Items to Pop-Up Menu Inherited from an Object's Ancestors" section. For the Workplace Shell to display the submenu as a conditional cascaded menu with the mini-push button and default selection, the menu's style and default selection must be set, as shown in the following sample code:

```
/*********************  Method Override  ****************************/

/* Add a new item to MyObject's pop-up menu  */
SOM_SCOPE BOOL32 SOMLINK MyObject_wpModifyPopupMenu
   (MyObject *somSelf, HWND hwndMenu, HWND hwndCnr, ULONG iPosition)

{
  .
  .

  MENUITEM mi;
  .
  .
```

```
  /* Get a handle to the New item submenu   */
  WinSendMsg(hwndMenu,
             MM_QUERYITEM,
             MPFROM2SHORT(ID_NEWITEMS),
             (MPARAM)&mi);
  hwndSubMenu = mi.hwndSubMenu;

  /* Query the menu's style  */
  ulstyle = WinQueryWindowULong(hwndSubMenu, QWL_STYLE);

  /* Add conditional cascaded capabilities to the existing menu style */
  ulstyle |= MS_CONDITIONALCASCADE;

  /* Set the menu style to include conditional cascaded capabilities  */
  WinSetWindowULong(hwndSubMenu, QWL_STYLE, ulstyle);

  /* Set the default selection in the submenu.  It must exist.  */
  WinSendMsg(hwndSubMenu,
             MM_SETDEFAULTITEMID,
             (MPARAM)ID_SUBITEM1, 0L);
  .
  .

}
```

Supporting User Selection of New Pop-Up Menu Items

When a class defines new actions for its pop-up menu, it must provide for the processing of the actions when the user selects the action. This is done by overriding the following pop-up menu methods:

Method	Description
wpMenuItemHelpSelected	Displays the help associated with class-specific pop-up menu item.
wpMenuItemSelected	Processes class-specific pop-up menu item.

Using the previous sample code, MyObject supports SubItem_1 by overriding wpMenuItemSelected, as shown in the following sample code:

```
/*******************  Method Override  *******************************/

/* Process input from the extra menu option that we added */
SOM_Scope void SOMLINK MyObject_wpMenuItemSelected
  (MyObject *self, HWND hwndFrame, ULONG MenuID)

{
  .
  .

  /* Which of our menu items was selected?  */
  switch( MenuId )
  {
    case ID_SUBITEM1:
        .
        .
    case ID_SUBITEM2:
        .
        .
    case ID_SUBITEM3:
        .
        .
    case ID_SUBITEM4:
        .
        .
    default:
       parent_wpMenuItemSelected
  }
```

```
}
```

MyObject can support help for new pop-up menu items by overriding wpMenuItemHelpSelected in a similar manner. These help methods are explained in Object Aid: Help Methods.

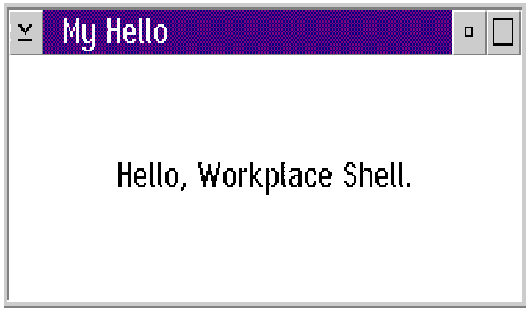
Sample Code for Pop-Up Menu Methods

This section illustrates a complete pop-up menu sample program. The source code for a sample Hello application is provided here to demonstrate how to customize a Workplace Shell pop-up menu. The application creates a new class, Hello, from the WPDataFile class with a modified pop-up menu, and then creates an instance of the new class called MyHello. An instance of the Hello class will be just like a WPDataFile object with the following exceptions:

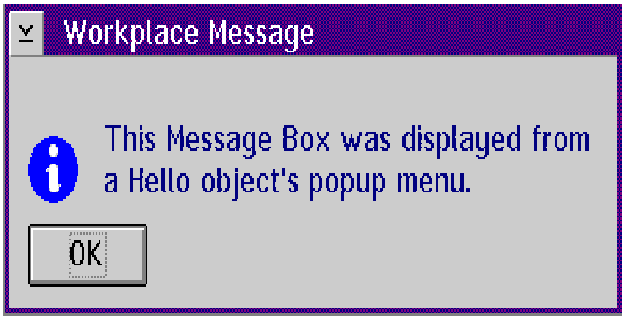
- The **Open** submenu will have a **Say Hello** item. MyHello's primary menu will have a **Message Box** item. The context menu will have the **Move** item removed. The following figure shows MyHello object and its pop-up menu:



- Selecting the **Open Say Hello** item or double clicking on the object will cause a window to display the text "Hello, Workplace Shell.", as shown in the following figure:



- Double clicking on the **Message Box** item will display a message box with information about this example, as shown in the following figure:



Pop-Up Menu Application Sample Code

The pop-up menu application includes the following files:

File Name	Description
HELLO.IDL	Class interface definition
HELLO.C	Source code
HELLO.RC	Resource code
HELLO.MAK	Hello make file for building the application

Following these files are three REXX files that register and deregister the new class with the Workplace Shell, and create an instance of the class.

Class Definition File for Hello

The following sample illustrates the class interface definition file (IDL):

```
//# Include the class definition file for the parent classes

#include <wpdataf.idl>
#include <somcls.idl>

interface M_Hello;   // Forward reference to metaclass

//#****************************************************************************
//#   Define the new class
//#****************************************************************************

interface Hello : WPDataFile
{
//
// CLASS: Hello
//
// CLASS HIERARCHY:
//
//     SOMObject
//           WPObject
//                 WPFileSystem
//                       WPDataFile
//                             Hello
//
// DESCRIPTION:
//
//    This is the sample class to demonstrate Workplace Shell menu
//    customization.
//
```

```
#ifdef __SOMIDL__
implementation
{
    //# Class modifiers
    externalstem   = hello;
    local;
    externalprefix = hlo_;
    majorversion   = 1;
    minorversion   = 2;

    filestem       = hello;
    metaclass      = M_Hello;
    callstyle      = idl;
    dllname        = "hello.dll";

    passthru C_h   =  ""
    "#define INCL_WIN"
    "#include <os2.h>"
    ""
    "/*"
    " *   The following user-defined pop-up menu items (ID_xxx) should"
    " *   be higher than WPMENUID_USER."
    " *"
    " *   The ID_OPENHELLO will become a submenu of the system's pop-up"
    " *   open menu id, WPMENUID_OPEN."
    " */"
    ""
    "#define ID_OPENHELLO   (WPMENUID_USER+1)  /* Menus to be added  */"
    "#define ID_MSGBOX      (WPMENUID_USER+2)"
    "#define IDM_OPENHELLO  (WPMENUID_USER+3)  /* Submenus of added menus */"
    "#define IDM_MSGBOX     (WPMENUID_USER+4)"
    "#define ID_FRAME       3000               /* Hello window frame ID */"
    ""
    "// The ID of the Hello window view. This is returned by the override"
    "// of wpclsQueryDefaultView to specify that the default view"
    "// is to open the Hello window. This ID MUST be the same as the ID of"
    "// the open menu item."
    ""
    "#define OPEN_HELLO     ID_OPENHELLO";

//#****************************************************************************
//#   Define overridden methods
//#****************************************************************************

    wpFilterPopupMenu: override;

// METHOD: wpFilterPopupMenu
//
// DESCRIPTION: Remove any menu items from the context
//              menu that don't apply.
//
// HOW TO OVERRIDE: No restrictions.
//
// NOTES:
//
//   This method is called when an objects' context menu has been requested
//   (i.e. right click or S-F10 on an object) before displaying the menu. It
//   is called before wpModifyPopupMenu.

    wpModifyPopupMenu: override;

// METHOD: wpModifyPopupMenu
//
// DESCRIPTION:  Add the menu items to the context menu.
//
// NOTES:
//
//   This method is called when an objects' context menu has been requested
//   (i.e. right click or S-F10 on an object) before displaying the menu. It
//   is called after wpFilterPopupMenu.

    wpMenuItemSelected: override;

// METHOD: wpMenuItemSelected
//
// DESCRIPTION: Processes input from the menu options that was added.
//
// NOTES:
//
//   This method is called when an item in an objects' context menu has been
//   selected.

    wpOpen: override;
```

```
// METHOD: wpOpen
//
// DESCRIPTION: Opens the Hello Workplace Shell window.
//
// NOTES:
//
//   This method is called when a new open view of an object is needed,
//   i.e. when an item in an object's Open submenu has been selected or
//   when an object is double clicked on.  This is equivalent to selecting
//   the default item in the object's Open submenu.
//
//   wpOpen should always open a new view. The Workplace Shell actually calls
//   wpViewObject first when an object is double clicked on or an open
//   submenu item is selected. wpViewObject will then call wpOpen if there
//   is currently no open view of the selected Open submenu item or if
//   mutliple concurrent views of the object are enabled.

};   /* End implementation */
#Endif /* __SOMIDL__ */
};

//#****************************************************************************
//#   Define the metaclass
//#****************************************************************************

interface M_Hello: M_WPDataFile
{

//#****************************************************************************
//#   Define metaclass methods
//#****************************************************************************

   HMODULE  clsQueryModuleHandle();

// METHOD: clsQueryModuleHandle
//
// DESCRIPTION:
//
//   Returns the module handle of this class.  If this is the
//   first invocation, DosQueryModuleHandle is called to save the handle
//   for future invocations.
//
// RETURN:
//
//   0              Unsuccessful
//   non-zero       Module handle
//
// NOTES:
//
//   This method is called when a new open view of an object is needed,
//   i.e. when an item in an object's Open submenu has been selected or
//   when an object is double clicked on.  This is equivalent to selecting
//   the default item in the object's Open submenu.
//

//#****************************************************************************
//#   Define metaclass data
//#****************************************************************************

   attribute HMODULE hmod;

#ifdef __SOMIDL__
implementation
{
    releaseorder: clsQueryModuleHandle;

    //# Class modifiers
    externalstem   = hello;
    local;
    externalprefix = hloM_;
    majorversion   = 1;
    minorversion   = 2;
    filestem       = hello;
    callstyle      = oidl;
    dllname        = "hello.dll";

//#****************************************************************************
//#   Define overridden metaclass methods
//#****************************************************************************

   wpclsQueryDefaultView: override;
```

```
// METHOD: wpclsQueryDefaultView
//
// DESCRIPTION: Returns the default view for a new instance
//              of this object.
//
// REMARKS:
//
//   Return the ID of the default view, OPEN_HELLO.
//   This ID must be the same as the ID of the open
//   submenu item, ID_OPENHELLO.
//
// NOTES:
//
//   This method is called to determine which view to open when an
//   object of class Hello is double clicked on.

   wpclsQueryTitle: override;

// METHOD: wpclsQueryTitle
//
// DESCRIPTION:  Return the string "Hello Workplace Shell".
//
// NOTES:
//
//   This method is called by the Workplace Shell to determine the object
//   title when an object of class Hello is created.
//
  }; /* End implementation */
#endif /* __SOMIDL__ */
};
```

Source Code for Hello

The following sample illustrates the source code (C):

```
/*
 *  This file was generated by the SOM Compiler and Emitter Framework.
 *  Generated using SOM Emitter emitctm: 2.40
 */

#ifndef SOM_Module_hello_Source
#define SOM_Module_hello_Source
#endif
#define Hello_Class_Source
#define M_Hello_Class_Source
#define INCL_DOS

#include "hello.ih"

MRESULT EXPENTRY ClientWinProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2);

// Structure to pass the object pointer and
// use list item to the window procedure
typedef struct
{
   WPObject *self;     /* Pointer to the object    */
   USEITEM  UseItem;   /* Use list item header     */
   VIEWITEM ViewItem;  /* Open view use list item  */
} WINDOWDATA;
typedef WINDOWDATA *PWINDOWDATA;

/*
 * METHOD: wpFilterPopupMenu
 *
 * DESCRIPTION: Removes any menu items from the context menu that don't apply.
 *
 * HOW TO OVERRIDE: No restrictions.
 *
 * NOTES:
 *
 *   This method is called when an object's context menu has been requested
```

```
 *   (i.e. right click or S-F10 on an object) before displaying the menu. It
 *   is called before wpModifyPopupMenu.
 */
SOM_Scope ULONG  SOMLINK hlo_wpFilterPopupMenu(Hello somSelf,
                                               ULONG ulFlags,
                                               HWND hwndCnr,
                                               BOOL fMultiSelect)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","hlo_wpFilterPopupMenu");

    // Return the parent class' menu bitstring minus the "move bit"
    return (parent_wpFilterPopupMenu(somSelf,
                                     ulFlags,
                                     hwndCnr,
                                     fMultiSelect) & ~CTXT_MOVE);
}

/*
 * METHOD: wpModifyPopupMenu
 *
 * DESCRIPTION: Adds the menu items to the context menu.
 *
 * NOTES:
 *
 *   This method is called when an object's context menu has been requested
 *   (i.e. right click or S-F10 on an object) before displaying the menu. It
 *   is called after wpFilterPopupMenu.
 */

SOM_Scope BOOL  SOMLINK hlo_wpModifyPopupMenu(Hello somSelf,
                                              HWND hwndMenu,
                                              HWND hwndCnr, ULONG iPosition)

{
    HMODULE hmod;
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","hlo_wpModifyPopupMenu");

    /* Get the module handle of the .DLL */
    hmod = _clsQueryModuleHandle(_Hello);

    /*
     *   Put the "Say Hello" submenu item in the Open submenu
     */
    _wpInsertPopupMenuItems(somSelf,          // Object
                            hwndMenu,         // Main context menu
                            0,                // Position to insert menu at
                                              // 0 = top of menu
                            hmod,             // Module of menu to insert
                            ID_OPENHELLO,     // Menu to insert
                            WPMENUID_OPEN);   // Submenu to insert menu in

    /*
     *   Put the "Product Info" menu item at the end of the main context menu
     */
    _wpInsertPopupMenuItems(somSelf,          // Object
                            hwndMenu,         // Main context menu
                            iPosition,        // Position to insert menu at
                                              // iPosition = next available
                            hmod,             // Module of menu to insert
                            ID_MSGBOX,        // Menu to insert
                            0);               // Submenu to insert menu in
                                              // 0 = main context menu

    return (parent_wpModifyPopupMenu(somSelf,
                                     hwndMenu,
                                     hwndCnr,
                                     iPosition));
}

/*
 * METHOD: wpMenuItemSelected
 *
 * DESCRIPTION:  Processes input from the menu options that was added.
 *
 * NOTES:
 *
 *   This method is called when an item in an objects' context menu has been
 *   selected.
 */

SOM_Scope BOOL  SOMLINK hlo_wpMenuItemSelected(Hello somSelf,
```

```
                                        HWND hwndFrame,
                                        ULONG ulMenuId)

{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug("Hello","hlo_wpMenuItemSelected");

    switch(ulMenuId)
    {
       case IDM_OPENHELLO:
          /*
           *   Either the object was double clicked on or the Say Hello
           *   item on the Open submenu was selected.
           *
           *   We could call wpOpen here, but wpOpen will open a new view
           *   no matter what. wpViewObject only opens a new view if
           *   no open view exists or if mutliple concurrent views are enabled.
           *   Otherwise, it will give focus to the current open view.
           *
           *   wpViewObject should be used whenever possible instead of wpOpen
           *   to prevent creating multiple open views inadvertently.
           */
          _wpViewObject(somSelf,             // Object
                        NULLHANDLE,          // Workplace Shell
                                             // internal use only
                        OPEN_HELLO,          // View to open. This ID must be the
                                             // same as the menu ID ID_OPENHELLO
                        0);                  // parameter passed to wpOpen
          break;

       case IDM_MSGBOX:
       {
          // Message Box menu item was selected
          WinMessageBox(HWND_DESKTOP,
                        HWND_DESKTOP,
                        "Message box displayed from a Hello object's pop-up menu.",
                        "Workplace Message",
                        1,
                        MB_OK       |
                        MB_MOVEABLE |
                        MB_INFORMATION);
          break;
       }

       default:
          return parent_wpMenuItemSelected(somSelf, hwndFrame, ulMenuId);
    }
    return TRUE;  /* We processed it */
}

/*
 * METHOD: wpOpen
 *
 * DESCRIPTION:  Opens the Hello Workplace Shell window.
 *
 * NOTES:
 *
 *   This method is called when a new open view of an object is needed,
 *   i.e. when an item in an object's Open submenu has been selected or
 *   when an object is double clicked on.  This is equivalent to selecting
 *   the default item in the object's Open submenu.
 *
 *   wpOpen should always open a new view. The Workplace Shell actually
 *   calls wpViewObject first when an object is double clicked on or an Open
 *   submenu item is selected. wpViewObject will then call wpOpen if there
 *   is currently no open view of the selected Open submenu item or if
 *   mutliple concurrent views of the object are enabled.
 */

SOM_Scope HWND  SOMLINK hlo_wpOpen(Hello somSelf,
                                   HWND hwndCnr,
                                   ULONG ulView,
                                   ULONG param)

{
   /* HelloData *somThis = HelloGetData(somSelf); */
   HelloMethodDebug("Hello","hlo_wpOpen");

   switch (ulView)
   {
      case OPEN_HELLO:
      {
         // Request to open a Hello Workplace Shell window
```

```
        // Create a standard window
        HAB     hab;                        // Anchor block handle
        HWND hwndClient;                    // Client window handle
        HWND hwndFrame;                     // Frame window handle
        PWINDOWDATA pWindowData;
        ULONG flCreate;                     // Window creation flags
        hab = WinQueryAnchorBlock(HWND_DESKTOP);

        // Create the class, adding a user word to the window data to
        // anchor the object use list item for this open view
        WinRegisterClass(hab,
                         "hello",
                         ClientWinProc,
                         CS_SIZEREDRAW,
                         sizeof(*pWindowData));

        flCreate = FCF_SYSMENU       |
                   FCF_SIZEBORDER    |
                   FCF_TITLEBAR      |
                   FCF_MINMAX        |
                   FCF_SHELLPOSITION |
                   FCF_TASKLIST;

        hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
                                       WS_VISIBLE,
                                       &flCreate,
                                       "hello",
                                       _wpQueryTitle(somSelf),
                                       0L,
                                       0,
                                       ID_FRAME,
                                       &hwndClient);

        if (hwndFrame)
        {
           WinSetFocus(HWND_DESKTOP, hwndFrame);

           // Now that we have created an open view, add the view to the
           // object's use list. This use list is used by wpViewObject and
           // wpClose to check for existing open views.


           // Get storage for and initialize a use list item
           pWindowData = (PWINDOWDATA) _wpAllocMem(somSelf,
                                                   sizeof(*pWindowData),
                                                   NULL);
           memset((PVOID) pWindowData, 0, sizeof(*pWindowData));
           pWindowData->self            = somSelf;
           pWindowData->UseItem.type    = USAGE_OPENVIEW;
           pWindowData->ViewItem.view   = OPEN_HELLO;
           pWindowData->ViewItem.handle = hwndFrame;

           // Add the use list item to the object's use list
           _wpAddToObjUseList(somSelf, &(pWindowData->UseItem));

           // Save the pointer to the use item in the window words so that
           // the window procedure can remove it from the list when the window
           // is closed
           WinSetWindowPtr(hwndClient, QWL_USER, pWindowData);
        }
        return hwndFrame;
     }

     // Some other view was requested. Pass the request on to the
     // parent (WPDataFile) class
     default:
        return (parent_wpOpen(somSelf, hwndCnr, ulView, param));
  }   /* end switch (ulView) */
}

/*
 * METHOD: clsQueryModuleHandle
 *
 * DESCRIPTION:
 *
 *   Returns the module handle of this class.  If this is the
 *   first invocation, DosQueryModuleHandle is called to save the handle
 *   for future invocations.
 *
 * RETURN:
 *
 *   0             Unsuccessful
 *   non-zero      Module handle
```

```
 *
 * NOTES:
 *
 *   This method is called when a new open view of an object is needed,
 *   i.e. when an item in an object's Open submenu has been selected or
 *   when an object is double clicked on. This is equivalent to selecting
 *   the default item in the object's Open submenu.
 */

SOM_Scope HMODULE  SOMLINK hloM_clsQueryModuleHandle(M_Hello somSelf)
{
    M_HelloData *somThis = M_HelloGetData(somSelf);
    M_HelloMethodDebug("M_Hello","hloM_clsQueryModuleHandle");

    //  Make sure we already have the module handle
    if (_hmod == NULLHANDLE)
    {
       zString zsPathName;
       somId   Id;

        //   Retrieve registered path name of the module (DLL)
        //   and query the module handle
       Id = somIdFromString("M_Hello");
       zsPathName = _somLocateClassFile(SOMClassMgrObject,
                                        Id,
                                        M_Hello_MajorVersion,
                                        M_Hello_MinorVersion);
       SOMFree(Id);
       DosQueryModuleHandle(zsPathName, &_hmod);
    }
    return (_hmod);
}

/*
 * METHOD: wpclsQueryDefaultView
 *
 * DESCRIPTION: Returns the default view for a new instance of this object.
 *
 * REMARKS:
 *
 *   Return the ID of the default view, OPEN_HELLO. This ID must be the
 *   same as the ID of the Open submenu item, ID_OPENHELLO.
 *
 * NOTES:
 *
 *   This method is called to determine which view to open when an object
 *   of class Hello is double clicked on.
 */

SOM_Scope ULONG  SOMLINK hloM_wpclsQueryDefaultView(M_Hello somSelf)
{
    /* M_HelloData *somThis = M_HelloGetData(somSelf); */
    M_HelloMethodDebug("M_Hello","hloM_wpclsQueryDefaultView");

    return OPEN_HELLO;
}

/*
 * METHOD: wpclsQueryTitle
 *
 * DESCRIPTION: Returns the string "Hello Workplace Shell".
 *
 * NOTES:
 *
 *   This method is called by the Workplace Shell to determine the object
 *   title when an object of class Hello is created.
 */

SOM_Scope PSZ  SOMLINK hloM_wpclsQueryTitle(M_Hello somSelf)
{
    // M_HelloData *somThis = M_HelloGetData(somSelf);
    M_HelloMethodDebug("M_Hello","hloM_wpclsQueryTitle");

    return "Hello Workplace Shell";
}

/*
 * FUNCTION: ClientWinProc
 *
 * DESCRIPTION: Window procedure for the Hello, Workplace Shell window.
 */

MRESULT EXPENTRY ClientWinProc (HWND hwnd,
```

```
                                ULONG msg,
                                MPARAM mp1,
                                MPARAM mp2)

{
    BOOL  fSuccess;  // Return from function
    switch(msg)
    {
        case WM_PAINT:
        {
            HPS     hps;               // Cached PS
            RECTL   rcl;               // Window rectangle
            CHAR    sz[25];            // String

            hps      = WinBeginPaint (hwnd , (HPS)0, NULL);
            fSuccess = GpiErase (hps);

            WinQueryWindowRect (hwnd, &rcl);
            strcpy (sz, "Hello, Workplace Shell.");
            WinDrawText(hps,
                        strlen(sz),
                        sz,
                        &rcl,
                        CLR_BLACK,
                        CLR_WHITE,
                        DT_CENTER |
                        DT_VCENTER);

            fSuccess = WinEndPaint (hps);
            return (MRESULT) NULL;
        }

        case WM_CLOSE:
        {
           // Get the object pointer and the use list item from the window
           PWINDOWDATA pWindowData = (PWINDOWDATA) WinQueryWindowPtr(hwnd, QWL_USER);

           // Remove this window from the object's use list
           _wpDeleteFromObjUseList(pWindowData->self,&pWindowData->UseItem);

           // Free the use list item. Note that there is no need to supply a
           // length since this storage was allocated with wpAllocMem
           _wpFreeMem(pWindowData->self,(PBYTE)pWindowData);

           // Destroy the window and return
           WinDestroyWindow(WinQueryWindow(hwnd, QW_PARENT));
           return (MRESULT) NULL;
        }

        default:
           return WinDefWindowProc(hwnd, msg, mp1, mp2);
    }
}
```

Resource Code for Hello

The following sample illustrates the resource code (RC):

```
#include "hello.ih"

MENU ID_OPENHELLO LOADONCALL MOVEABLE DISCARDABLE
BEGIN
   MENUITEM "~Say Hello", IDM_OPENHELLO
END

MENU ID_MSGBOX LOADONCALL MOVEABLE DISCARDABLE
BEGIN
   MENUITEM "~Message Box", IDM_MSGBOX
END
```

Make File for Hello

The following sample illustrates the make file (MAK):

```
all:        hello.dll

hello.dll:  hello.obj hello.res
            link386 /co /noi /noe /m hello,hello.dll,,somtk pmwp.lib,hello.def;
            rc hello.res hello.dll

hello.obj:  hello.c
            icc /c /Ti /Kb /Gm /Ss /Ge- hello.c

hello.res:  hello.rc
            rc -r hello.rc
```

REXX Files for Hello

The following shows the REXX command file for registering the Hello class:

```
/* */
call RxFuncAdd "SysLoadFuncs", "RexxUtil", "SysLoadFuncs"
call SysLoadFuncs
say  SysRegisterObjectClass("Hello", "C:\OS2\DLL\HELLO.DLL")
```

The following shows the REXX command file for creating an instance of the Hello class:

```
/* */
call RxFuncAdd "SysLoadFuncs", "RexxUtil", "SysLoadFuncs"
call SysLoadFuncs
say  SysCreateObject("Hello",,
                     "My Hello",,
                     "<WP_DESKTOP>",,
                     "OBJECTID=<MY_HELLO>")
```

The following shows the REXX command file for deregistering the Hello class:

```
/* */
call RxFuncAdd "SysLoadFuncs", "RexxUtil", "SysLoadFuncs"
call SysLoadFuncs
say  SysDeregisterObjectClass("Hello")
```

REXX Utility Workplace Shell Functions

REXX combines the simplicity of a programming language such as BASIC with features that exist in more powerful languages such as C, PASCAL, or PL/1. Part of the success of REXX has been due in part to having been selected as the Systems Application Architecture Programming Language (SAA/PL). The purpose is to provide a common look and feel across all operating systems. This is important because CUA is the standard that the Workplace Shell uses which is also part of SAA. REXX provides the facility for the Workplace Shell to create, modify, and delete objects. This chapter describes the REXX functions and how to use them in Workplace Shell applications.

About REXX Utility Functions

The power of REXX is often overlooked by many people as to how much benefits one can receive by using a command based language. REXX which stands for restructured extended executor was created to offer an easy way to ask the operating system to perform a special task without writing hundreds or sometimes thousands of lines of computer code to do the same task. The simplicity of REXX is by far its strongest point along with its ease of use. REXX provides the facility for the Workplace Shell to create, modify, and delete objects in a very simplistic way.

REXXUTIL is a dynamic link library (REXXUTIL.DLL) that provides specific functionality to the Workplace Shell, which allows manipulating Workplace Shell classes as well as objects. A complete description of the commands can be found in the *OS/2 Command Reference* or the online version which is located in the OS/2 Information folder.

To use a REXXUTIL function in a REXX program, you must first register the function using RxFuncAdd. However, you have the choice of either registering one function using RxFuncAdd or you can register all functions by first registering SysLoadFuncs using RxFuncAdd. The following example shows how to register a single function:

```
call RxFuncAdd 'SysSetObjectData', 'RexxUtil', 'SysSetObjectData'
```

The function SysSetObjectData is the only function in this case that is registered by RxFuncAdd. The following example shows how to register all of the functions using RxFuncAdd:

```
call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
call SysLoadFuncs
```

Using SysLoadFuncs allows all of the OS/2 sessions to use the REXXUTIL functions.

Using REXX Utility Functions

There are useful functions that pertain to the object oriented Workplace Shell. REXXUTIL provides useful functions that allow the creation, destruction, and the altering of the properties of objects. These functions are listed as follows:

- SysCreateObject
- SysDestroyObject
- SysSetObjectData
- SysSetIconData
- SysRegisterObjectClass
- SysDeregisterObjectClass
- SysQueryClassList
- SysGetEa
- SysPutEa
- SysIni

Creating Objects

The following figure shows the syntax of SysCreateObject:

```
Syntax:  SysCreateObject(class_name,
                         title,
                         location,
                         <setup_string>,
                         <option>)

where

class_name       Name of the Workplace Shell class of which the object
                 is a member of, such as WPFolder.
```

```
title            New object's title.
location         Either an object ID, such as <WP_DESKTOP>, or a fully
                 qualified path name.
setup_string     A series of keynames that change the behavior of the
                 object.  The keynames are separated by a semicolons
                 and values are separated by commas.
option           An action to be taken if the object class already
                 exists.  The action could be FAIL, REPLACE, or UPDATE.
```

The following sample code creates two objects, one of the class WPFolder and one of the class WPProgram. The program object MyProgram is inserted into the Folder MyFolder. The location specified by MyProgram is the same as the object ID for MyFolder.

```
/* */
call RxFuncAdd "SysLoadFuncs", "rexxutil", "SysLoadFuncs"
call SysLoadFuncs

say SysCreateObject("WPFolder",,
                    "MyFolder",,
                    "<WP_DESKTOP>",,
                    "OBJECTID=<Prefix_MyFolder>")

say SysCreateObject("WPProgram",,
                    "MyProgram",,
                    "<Prefix_MyFolder>",,
                    "EXENAME=C:\PGM\PGM.EXE;ASSOCFILTER=*.TXT" ,,
                    "OBJECTID=<Prefix_MyProgram>")
```

Destroying Objects

The following figure shows the syntax of SysDestroyObject:

```
Syntax:  SysDestroyObject(object_name)

where

object_name     Name of the object that is either the object's ID
                assigned to the object when it was created, or the
                fully qualified path name of the object to be destroyed.
```

The following sample code shows how to destroy and recreate the color palette objects. To destroy the objects first, the setup string must be modified to allow the color palette objects to be destroyed. The setup string NODLETE = NO must be changed to NODELETE = YES, so that the object can be destroyed.

```
/***/
call RxFuncAdd 'SysLoadFuncs','RexxUtil','SysLoadFuncs'
call SysLoadFuncs

/* ColorObjectId  = "<WP_CLRPAL>" */
ColorObjectId1 = "<WP_LORESCLRPAL>"
ColorObjectId2 = "<WP_HIRESCLRPAL>"

ColorObjectLocation = "<WP_CONFIG>"
call SysSetObjectData ColorObjectId1, "NODELETE=NO"
call SysSetObjectData ColorObjectId2, "NODELETE=NO"

/* Call SysDestroyObject ColorObjectId */
call SysDestroyObject ColorObjectId1
call SysDestroyObject ColorObjectId2

rc = SysCreateObject('WPColorPalette',,
        'Solid Color Palette',,
        '<WP_CONFIG>',,
        'AUTOSETUP=LORES;NODELETE=NO;OBJECTID='ColorObjectId1,,
        'update');
```

```
rc = SysCreateObject('WPColorPalette',,
        'Mixed Color Palette',,
        '<WP_CONFIG>',,
        'AUTOSETUP=HIRES;NODELETE=NO;OBJECTID='ColorObjectId2,,
        'update');
```

Updating Objects

The following figure shows the syntax of SysSetObjectData:

```
Syntax:  SysSetObjectData(object_name, setup_string)

where

object_name      Name of the object that is either the object's ID
                 assigned to the object when it was created, or the
                 fully qualified path name of the object to be
                 destroyed.
setup_string     A series of keynames that change the behavior of the
                 object.  The keynames are separated by a semicolons
                 and values are separated by commas.
```

Each object's characteristics are modified by the setup string information that is passed to the object. The following sample code shows what types of setup information can be used to modify objects' characteristics:

```
/* */
call RxFuncAdd "SysLoadFuncs", "rexxutil", "SysLoadFuncs"
call SysLoadFuncs

say SysSetObjectData("<WP_TEMPS>",,"NODELETE=NO");
say SysSetObjectData("C:\DESKTOP",,"OBJECTID=<WP_DESKTOP>")
say SysSetObjectData("<Prefix_MyProgram",,"OPEN=DEFAULT")
say SysSetObjectData("<WP_DESKTOP>",,"TITLE=Comany Name Desktop")
say SysSetObjectData("c:\pgm\pgm.exe",,"ASSOCTYPE=Plain Text, C Code")
```

Registering and Deregistering Objects

The following figure shows the syntax of SysRegisterObjectClass:

```
Syntax:  SysRegisterObjectClass(class_name, module_name)

where

class_name       Name of the Workplace Shell class of which the object
                 is a member, such as WPFolder.
module_name      Name of the DLL from which the object is loaded from.
```

The following figure shows the syntax of SysDeregisterObjectClass:

```
Syntax:  SysDeregisterObjectClass(class_name)

where

class_name     Name of the Workplace Shell class to deregister.
```

In the following sample code REXX registers the class MyClass from the DLL defined with the fully qualified path to the system. All of the properties of the class MyClass are loaded through MYCLASS.DLL. Deregistering the class simply involves passing the class_name to successfully free the class from the system.

```
/* */
call RxFuncAdd "SysLoadFuncs", "rexxutil", "SysLoadFuncs"
call SysLoadFuncs

say SysRegisterObjectClass("MyClass", "c:\pgm\myclass.dll")
say SysDeregisterObjectClass("MyClass")

call SysQueryClassList "list."

do i=1 to list.0
  say 'class ' i  ' is ' list.i
end
```

Getting Extended Attributes

The following figure shows the syntax of SysGetEA:

```
Syntax:  SysGetEA(file, name, variable)

where

file         File containing the extended attribute.
name         Name of the extended attribute, for example .ICON.
variable     Name of the REXX variable containing the extended
             attribute value.
```

The Workplace Shell uses the following extended attribute headers:

Extended Attribute	Description
.ASSOCTABLE	Enables an application to indicate the type, extension, and icon for the data file that it recognizes.
.CLASSINFO	Specifies the class type of the class object.
.ICON	Specifies the icon that represents the file.
.LONGNAME	Stores or restores a long file name that is copied to media (floppy drive, hard drive, tape) not supporting long file names.
.TYPE	Specifies the file type of the file object, such as plain text (TXT), bitmap (BMP), or executable (EXE).

The following sample code fragments stores the file's extended attribute icon data to a temporary file. The icon data file is then modified by using the icon editor. Once saved, the icon data extended attribute is updated in the file.

```
/* */
Arg Filename
call RxFuncAdd "SysLoadFuncs", "rexxutil", "SysLoadFuncs"
call SysLoadFuncs

TempName = 'TEMP$.ICO'
If (Stream (TempName, 'c', 'query exists') <> ' ') then
  'ERASE ' TempName     /* If the TempFile exists, erase it */

If SysGetEA(Filename, '.icon', 'ICON') = 0 then
  do   /* Load the icon extended attribute to the TempFile */
    Parse Var ICON AInfo 5 ICON
    do I = 1 to Length(ICON)
      Rc = Charout(TmpName, Substr(Icon, I, 1))
```

```
    end
  If Length(Icon) \= 0 then Rc = Charout(TempName)
  end

say
say 'The icon data for the file ' Filename ' will now be open '
say 'into the icon editor.'
say 'You can update the icon.'
say 'If this is a brand new icon, you will be prompted to '
say 'save the data to a file.  Save it to ' TempName '.'
say 'When you are ready, press the Enter key.'

Rc=SysGetKey()

'@ICONEDIT 'TempName                  /* Start the icon editor */
Rc = SysSetIcon(Filename, TempName)   /* Update the icon data  */
                                      /* extended attributes   */
'@DEL 'TempName
return
```

Extended Attributes and the Workplace Shell

Applications not written for the Workplace Shell need to be aware of how .LONGNAME and .ASSOCTABLE extended attributes are used by the Workplace Shell.

In the Workplace Shell, the user can edit the name of an object. When an object is a file system object, the Workplace Shell renames the file object to match the name the user has entered. When the file system object resides on an HPFS disk, the new file name can have a long name and can accommodate whatever the user has entered. When the file system object resides on a FAT disk, the new file name must be no longer than eight characters. If the user has entered a name longer than eight characters, the Workplace Shell uses the first eight characters to rename the object and places the entire title in the .LONGNAME extended attribute associated with the file system object. This means that the title of a file object is the .LONGNAME extended attribute or the file name, if no .LONGNAME exists.

An .ASSOCTABLE extended attribute contains information that associates data files with the applications that create them or that know how to use them. Applications that a data file has been associated with appear in the list of **Open** actions for the data file. This means that opening the file is equivalent to starting the application that creates or modifies that file. The application is passed the name of the file as a command-line parameter.

.ASSOCTABLE extended attributes are defined in an application's resource file, as shown in the following example:

```
ASSOCTABLE assoctable-id [load-option] [mem-option]
BEGIN
  association-name, file-match-string [,extended-attribute-flag] [,icon-filename]
END
```

When an application that defines an .ASSOCTABLE is installed in the Workplace Shell, the Workplace Shell automatically creates object templates for each type of data file that has been associated with the application.

Updating Extended Attributes

The following figure shows the syntax of SysPutEA:

```
Syntax:  SysPutEA(file, name, variable)

where

file         File name containing the extended attribute.
name         Name of the extended attribute, for example, .ICON.
variable     New value of the extended attribute.
```

The following sample code fragments creates a new extended attribute called MYATTRIBUTE and adds a comment to it:

```
/* */
call RxFuncAdd "SysLoadFuncs", "rexxutil", "SysLoadFuncs"
call SysLoadFuncs

SysPutEA('c:\myattr.cmd', MYATTRIBUTE, 'This is my attribute!!!!!')
```

Retrieving All Registered Object Classes

The following figure shows the syntax of SysQueryClassList:

```
Syntax:  SysQueryClassList(stem)

where

stem     Name of a stem variable in which all registered classes
         are placed.
```

The following sample code retrieves a complete list of registered objects:

```
/* */
call RxFuncAdd 'SysLoadFuncs', 'rexxutil', 'SysLoadFuncs'
call SysLoadFuncs

SysQueryClassList('Class_list.')
do cnt = 1 to Class_list.0
  say Class_list.cnt
end
```

Setting, Querying, and Deleting Keys and Applications

The following figure shows the syntaxes of SysIni:

```
Syntax - Mode 1:  Setting single key value.

    SysIni([inifile], app, key, val)

Syntax - Mode 2:  Querying single key value.

    SysIni([inifile], app, key)

Syntax - Mode 3:  Deleting a single key.

    SysIni([inifile], app, key, 'DELETE:')

Syntax - Mode 4:  Deleting an application and all associated keys.

    SysIni([inifile], app, ['DELETE:'])

Syntax - Mode 5:  Querying names of all keys associated
                  with a certain application.

    SysIni([inifile], app, 'ALL:', 'stem')

Syntax - Mode 6:  Querying names of all applications.
```

```
    SysIni([inifile], 'ALL:', 'stem')

where

inifile     Name of the INI file that you work with.  This parameter
            should be a file specification, or one of the following:

            USER       User INI file (usually C:\OS2\OS2.INI).
                       This is the default.
            SYSTEM     System INI file
                       (usually C:\OS2\OS2SYS.INI).
            BOTH       For querying invocations, both the user and
                       system INI files are searched.  For setting
                       invocations, the user INI file is written to.

app         Application name or some other meaningful value with which
            you would like to store keywords.
key         Name of a keyword that is used to hold data.
val         Value to associate with the keyword of the specified
            application.
stem        Name of the stem variable to store the resultant
            information in.  STEM.0 will be set equal to the number of
            elements.
```

The following sample code shows how you can use the SysIni function. The code saves the value entered by the user, displays it, and deletes it.

```
/***  Save the user-entered vlaue under the  ***/
/***  key 'NAME' of the 'MYAPP' application, ***/
/*** display it, then delete it.             ***/
pull name .
call SysIni , 'MYAPP', 'NAME', value  /* Save the value              */
say  SysIni(, 'MYAPP', 'NAME')        /* Query the value             */
call SysIni , 'MYAPP'                 /* Delete all MYAPP information */
exit
```

The following sample code displays the application name, keyname, and value of each line of the OS2.INI file:

```
/***  Display all OS2.INI file information to the screen  ****/
call rxfuncadd sysloadfuncs, rexxutil, sysloadfuncs
call sysloadfuncs
call SysIni 'USER', 'All:', 'Apps.'

if Result \= 'ERROR:' then
  do i=1 to Apps.0
    call SysIni 'USER', Apps.i, 'All:', 'Keys'

    if Result \= 'ERROR:' then
      do j=1 to Keys.0
        val = SysIni('USER', Apps.i, Keys.j)
        say left(Apps.i, 20) left(Keys.j, 20),
        'Len=x'''Left(d2x(length(val)),4) left(val, 20)
      end
    end
```

The following sample code fragments retrieves a complete list of object ID information:

```
/* */
call RxFuncAdd 'SysLoadFuncs', 'rexxutil', 'SysLoadFuncs'
call SysLoadFuncs

/* List of object IDs */
App='PM_Workplace : Location'
call RxFuncAdd 'SysLoadFuncs', 'rexxutil', 'SysLoadFuncs'
call SysLoadFuncs

call SysINI 'USER', App, 'All:', 'Keys'
if result \= 'Error:' then do
  call SysCls
  Say '';Say '';Say 'Listing of object ID information';Say '';
  parse value SysTextScreenSize() with row col
  j=row-10
```

```
  do i=1 to Keys.0
    if trunc(i/j)==i/j then
    do
      Say '';Say 'Press any key to display the next screen...'
      key=SysGetKey()
      call SysCls
      Say '';Say '';Say 'Listing of object ID information';Say '';
    end
    Say Keys.i
  end
end
else Say 'Error querying for' App
return
```

WorkPlace Shell Setup Strings

The setup strings are used to specify custom attributes for creating Workplace Shell objects. The keynames can be concatenated together in some cases for defining the object. The INI.RC file that is located in the OS/2 subdirectory contain all of the Workplace Shell install objects. These objects are comprised of the title, class_name, setup string, and object ID. The keynames in the setup string might have to be set to values such as ON, OFF, YES, or NO. Every class may define its own set of keynames and values. The *Workplace Shell Reference* contains the definition of keynames for all classes. All keynames for WPObject class are listed in Workplace Shell Setup Strings.

Workplace Shell and System Object Model

SOM is the object technology used for the implementation of the Workplace Shell. It provides a language neutral, object-oriented programming methodology that you can use to create Workplace Shell objects and bind them to the Workplace Shell. This chapter describes SOM's architecture, its features and environment. It describes its methods, classes, and finally provides an example of its workings. This chapter is not intended to be a SOM programming guide. It is, however, intended to serve as an introduction to SOM at a level necessary to write Workplace Shell applications.

About Workplace Shell and SOM

The *System Object Model* (SOM) is IBM's object-oriented programming model which provides you language-neutral object-oriented programming mechanisms, a set of run times for the support and management of objects, frameworks for the implementation of classes and objects, and tools for associated application development.

Workplace Shell is the OS/2 user interface and is implemented by using the SOM functions, methods, and structures. SOM's run-time support is what allows Workplace Shell applications to run on different levels of the operating system without having to be recompiled.

Object-oriented programming is a programming paradigm based on objects, which are programming constructs designed to reflect items in the real world. An *object* consists of both the data necessary to describe a real-world item and the functions necessary to describe the behavior of the item. This is in contrast to the structured programming model, which focuses on things that can be done to the data (the functions) and which treats the data only as something to be acted on. Objects bind together the data that describes an item and the functions that act on the data. The data is called the *object's state* . The functions that define the object's behavior are called *methods* . *Objects* are instances, or instantiations of a class. A *class* is a description of an object. It defines the data that represents the object's state and the methods that the object supports.

SOM is designed specifically to support the new, object-oriented paradigm, and to be usable with both procedural (non-object-oriented) languages and object-oriented languages. SOM is not a language-it is a system for defining, manipulating, and releasing class libraries. SOM is used to define classes and methods, while permitting you to choose a language for implementing these methods. Therefore, you will be able to use SOM quickly without having to learn a new language syntax. SOM consists of a run-time library and a set of utility programs that support building, externalizing, and manipulating software objects.

SOM objects are language-neutral. They can be defined in one programming language and used by applications or objects written in another programming language.

SOM Features

SOM offers three important object-oriented programming features:

- Encapsulation
- Inheritance
- Polymorphism.

Encapsulation

An object has an external or *public* view that prescribes how other objects or applications can interact with it. An object also has an internal or *private* view that prescribes how data and methods actually are implemented. Object implementation is hidden or *encapsulated* from the public view.

You can externalize as much of an object's definition as they choose. However, you should carefully consider what to choose to externalize. Published methods and instance variables become a permanent part of an object's interface. Unnecessary externalization of an object's definition might compromise future compatibility.

SOM permits changes to an object's internal implementation without affecting the compatibility of resulting binaries. This means that applications using SOM objects will not require recompilation when the SOM object definitions change. Full forward, binary compatibility can be retained when:

- Adding new methods
- Adding, changing, or deleting unpublished instance variables, provided that old methods still are supported
- Inserting new classes above your class in the inheritance hierarchy
- Relocating methods upward in the class hierarchy.

Inheritance

Inheritance is the derivation of new child classes from existing parent classes. Child classes inherit the characteristics of their parent classes. This means that methods defined for a parent class automatically are defined for the child class. In addition to those they have inherited, child classes also can add their own characteristics. This means that child classes can define new behavior in terms of new methods. These child classes also are known as *subclasses* .

Polymorphism

Polymorphism , basically, is many implementations of the same method for two or more classes of objects. This is known in SOM as *method overrides* , or *override resolution* . SOM supports various types of polymorphism, so it readily can be mapped into different object-oriented languages.

As child classes are derived from parent classes, inherited methods can be overridden. For example, suppose that ClassB is a child of ClassA, as illustrated in the following figure:

```
Definition of ClassA                    Definition of ClassB

MethodA:                                MethodA:
  Print something           `             Print something else
```

MethodA is one of the methods defined for ClassA. It also is one of the methods defined for ClassB and is inherited from ClassA. In SOM,

MethodA can be overridden in the definition of ClassB to do different things for ClassB. MethodA, therefore, is defined for both ClassA and ClassB but is implemented differently for ClassA and ClassB. This means that method resolution requires the name of the method as well as the object being acted upon.

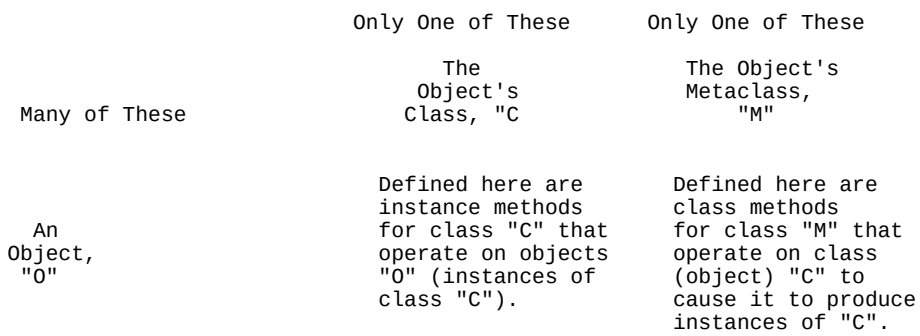
SOM Run-Time Environment

The SOM run-time environment contains the basic data structures and functions that are used to define, create, and manage classes and objects in terms of other classes.

Note: OS/2 operating system Version 3 includes SOM 2.1 run time and Workstation DSOM. The OS/2 Version 3 Toolkit contains a subset of the SOMobjects* 2.1 Toolkit.

Classes are generic definitions of sets of objects and their behaviors. Classes are defined at compilation time. *Class objects* are the SOM run-time implementation of SOM classes. Because the terms class and class object refer to the same thing, but in different contexts (compilation time and run time), they can be used interchangeably. Objects are created dynamically during run time. Objects are instances of classes. The methods that an object responds to are referred to as *instance methods* , because any object instance can perform them. An object's instance methods are defined in its class definition, and cannot be used unless an object instance already exists. Object instances are created by methods that operate on the class object to make it produce an object instance. Class methods that create object instances are called *factory methods* or *constructors* . SOM classes that define factory methods for classes are called *metaclasses* . Metaclasses are classes of classes. A class object is an instance of its metaclass.

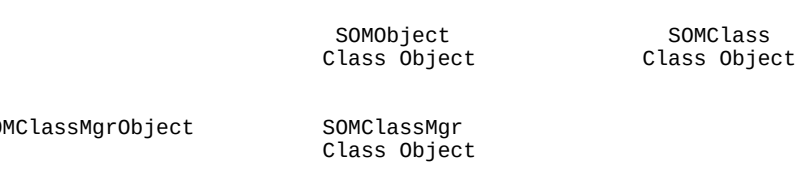
The relationship between objects, classes, and metaclasses is shown in the following figure.



The SOM environment can be created automatically or explicitly within any process that uses it. SOM supplies three classes, as shown in the following table. Classes that make up the SOM run-time environment are packaged with the operating system in SOM.DLL.

Object	Description
SOMClass	Root class for all SOM metaclasses.
SOMClassMgr	Class for SOMClassMgrObject.
SOMObject	Root class for all SOM classes.

SOMObject defines the essential behavior common to all SOM objects. All SOM classes are subclasses of SOMObject. SOMClass defines the essential behavior common to all SOM class objects. SOMClass is a subclass of SOMObject and is the metaclass of the SOMObject class. By definition, SOMClass is its own metaclass. SOMClassMgr is the class definition for the SOMClassMgrObject that is created during SOM initialization. During SOM initialization, four objects are created, as shown in the following figure. Three of these objects are class objects.



The SOMClass class object provides constructors for SOMObject class objects and for the SOMClassMgr class object. SOMObject defines a set of methods common to all SOM objects. Because all classes are subclasses of SOMObject, they inherit the set of methods common to all SOM objects. SOMClassMgrObject is an instance of the SOMClassMgr class object. SOMClassMgrObject dynamically loads and unloads class libraries when referenced and tracks instances of class objects.

Creating SOM Classes

The first step in creating a SOM class is to define the class and its relationship to other classes. In SOM, a class is defined in a formal specification language, the SOM interface definition language (IDL). IDL has a language-neutral core (appropriate for any programming language) to which some minor extensions have been added to simplify programming in C. IDL supports C and C++ programming languages. The class definition is saved in an ASCII text file with an extension of .IDL.

The second step is to process the IDL file using the SOM compiler to produce a set of language-specific or use-specific binding files for the class. The binding files are then used to build class libraries that can be used by client applications; that is, applications that are to create subclasses or object instances.

Note: Hereafter, any information concerning SOM bindings is specifically geared toward the SOM C-language bindings, as distinguished from other SOM language bindings.

Interface Definition Language

The class definition file provides a complete description of a class, including its relationship to other classes, its instance data, and the methods that it supports and overrides. An interface definition language (IDL) file for the C language also describes information specific to building C-language binding files.

An IDL specification for a single class in a single .IDL file generally has the sections specified in the sample code illustrated in the following sample code:

```
//# Include directives

//# Type declarations

//# Constant declarations

//# Exception declarations

//# Interface declaration

interface class-name : parent-class
{
   //# Constant declarations

   //# Type declarations

   //# Exception declarations

   //# Attribute declarations

   //# Method declarations

   //# Implementation statement

   #ifdef __SOMIDL__
   implementation
   {
      //# Modifier statements

      //# Passthru statements

      //# Declarations for instance variables
   };
   #endif   // __SOMIDL__

//# Module declaration
};
```

Include Directive Statement

Include directives tell the SOM compiler where to find the interface definition language (IDL) files associated with the class. These would be the IDL files for the class' immediate parents, its metaclass, and for any ancestor class for which this class overrides one or more of its methods.

Type, Constant, and Exception Declarations

The declaration section allows for the defining of types, constants, and exception structures. Any declarations defined here (outside of the interface declaration section described below) are not transferred to the binding files by the SOM compiler.

Interface Declarations

The interface declaration section describes the interfaces available to the class. By default, all interfaces defined in this section are public and available to client programs. These include any new methods defined for the class, as well as attributes. Additional types, constants, and exception structures may also be defined in this section. Any declarations defined in this section are exported by the SOM compiler to the language binding files. To make parts of the interface private, surround them with the preprocessor commands as shown in the following sample code:

```
//# Two underscores (__) before and after the word PRIVATE
#ifdef __PRIVATE__

//# Private attributes. constants, types, exceptions, and methods here

#endif
```

Declaring attributes creates instance data plus two accessor methods:

- One to set the value of the attribute
- One to retrieve the value of the attribute.

Implementation Statement

The implementation statement in the IDL file specifies how the class will be implemented. This includes the following:

- Names for the binding files
- Names of inherited methods that will be overriden

- Version number for the class
- Metaclass (if any)
- Release order for the methods
- Other class information.

Passthru Statement

Passthru statements allow the class implementor to specify blocks of code that the SOM compiler will pass into the Header files it generates. It is recommended that only #include directives be exported to C and C++ programs. Declarations for other data should be written in SOM and exported by the SOM compiler itself.

Instance Variables

Instance variables are intended to be used only by the class' methods and not by client programs or subclasses' methods. Attributes should be used for data that is manipulated by client programs or subclassers.

Default Processing Order

The order in which the SOM compiler processes the attributes and methods described in the IDL file is critical for future compatibility when new attributes or methods might be introduced in the class definition. By default, when the SOM compiler processes the IDL file, it builds the internal data structures for the binding files based on the order in which the attributes and methods are described in the interface section of the IDL file. If, at a later date, new data or methods change the original ordering of methods for the class, the binding files and class library are built differently. Client applications then must be recompiled.

Release Order List

The default processing order for methods can be overriden, however, by including a release order list in the implementation section of the IDL file. The *release order list* directs the SOM compiler to process the method definitions in the order specified in the list, as opposed to the order in the interface section. New methods should always be added to the end of the release order list to preserve compatibility with client applications. This allows you to describe the methods in the interface section in any order. In particular, new methods may be inserted in the middle of the existing methods without affecting client applications.

Comments

An IDL file is made more readable by the use of comments. Several comment styles are supported by IDL, as shown in the following figure:

```
/************************************/
/* This is a comment that will appear  */
/* in the language binding files.      */
/************************************/

/*
 * This is a comment that will appear
 * in the language binding files.
 */

// This is a comment that will appear
// in the language binding files.

//# This is a comment that will not appear
//# in the language binding files.
```

Because IDL files are used to generate language bindings, comments must be strictly associated with particular elements so that they appear at the appropriate points in the output files. If comments are not placed as prescribed by the IDL syntax, they might not appear where you expect to see them.

Comments beginning with //# do not appear in any binding files. They can be placed anywhere in an IDL file. They can be used to comment out sections of an IDL file or to add historical or notational commentary.

SOM Macros, Functions, and Data

To effectively use the SOM C-language bindings, an understanding of the SOM naming conventions is needed. These are summarized in the following table:

Prefix	Use
som	Function and method names.
SOM	Data items.

For example, methods for the SOM objects provided by the SOM run time have the prefix, *som* . Constants, data structures, and pointers to functions, including macro names, have the prefix, *SOM* .

Next, and most importantly, you need to understand SOM macros. SOM macros are used by class implementers and by client programs to:

- Instantiate objects
- Access object variables
- Invoke object methods.

SOM macros shield you from the details and complexity of SOM data structures, method resolution, and function invocations. There are two types of SOM macros:

- Class-specific
- Non-class specific (or general).

Class-Specific SOM Macros

Class-specific macros resolve references to class methods, class functions, and class instance data. The macro name or macro parameters contain a class method name, a class function name, an instance variable name, or a class name. The SOM compiler automatically generates and places them in the class Header files. Class implementers can use them by including the class-implementation Header file (.IH) in their source programs. Client programs can use them by including the public class Header file (.H). Because the IH file includes the H file, the class macros defined in the H file are available to both class implementers and clients; and the ones defined in the IH file but not in the H file are available only to class implementers.

The simplest macro is the " _ " (underscore) macro. Object instance data can be referred to by preceding the name of the data element with an underscore character, as shown in the following figure:

```
return _var1;
```

Underscored-data-name macros are defined in the H public class Header file and are available to both class implementers and client programs. In addition, instance variables may be accessed by somThis variable_name, as shown in the following figure:

```
return somThis var1;
```

Object methods can be referred to similarly and invoked by preceding the method name with an underscore (_) character, as shown in the following figure:

```
/* method A, with parameters x and y, operates on object obj */
_methodA(obj, x, y)
```

Underscored-method-name macros also are defined in the H public class Header file and are available to both class implementers and client programs. When nonrelated classes independently define methods with the same name, their methods can be invoked with a variation of this macro. The method name is prefixed with the underscore (_) character and class name, as shown in the following figure:

```
#include "classa.h"
#undef _methodA
#include "classa.h"

/* methodA, defined for ClassA objects,    */
/* operates on ClassA instance object obj1 */
ClassA_methodA (obj1);

/* methodA, defined for unrelated ClassB objects, */
/* operates on ClassB instance object obj2        */
ClassB_methodA (obj2);
```

Class objects also can be referred to by preceding the class name with an underscore character. Underscored-class-name macros also are defined in the H public class Header file and are available to both class implementers and client programs.

Other class-specific macros are summarized in the following table:

Macros	Function
Access instance data	get_<instance variable>
	<classname>GetData
Instantiate objects	<classname>New
	<classname>Renew
Invoke methods	SOM_Resolve
	SOM_ResolveNoCheck
Invoke methods for each parent	<classname>_parents_<methodname>
Invoke parent methods	<classname>_parent_<parentclassname> _<methodname>
Trace methods	<classname>MethodDebug

Of the macros listed in the previous table, <classname>GetData and the parent method macros are defined in the IH file, but not in the H file. They are available only for class implementers.

A class-specific function also is defined in the H file associated with a class. This function, <classname>NewClass, creates the class object. It is invoked automatically when an object is instantiated through the <classname>New class macro.

General SOM Macros and Functions

SOM provides a set of non-class-specific, or general SOM macros and functions that support:

- ID manipulation
- Debugging
- Error handling
- Getting object information.

General SOM macros are defined in the SOM.H Header file. Because SOM.H is included in the class Header (H and IH) files, SOM macros are available to class implementers and client programs.

SOM ID Manipulation

IDs are numbers that uniquely represent strings. They can be used in SOM to identify method names, class names, and descriptors. Typically, they are used to provide a fast and efficient means of comparing the strings they represent. SOM provides a set of macros and functions, as shown in the following table, that can be used to manipulate SOM IDs.

Type	Interface
Functions	somBeginPersistentIds
	somEndPersistentIds
	somRegisterId
	somSetExpectedIds
	somTotalRegIds
	somUniqueKey
Macros	SOM_CheckID
	SOM_CompareIDs
	SOM_IDFromString
	SOM_StringFromID

Initially an ID is a pointer to a string. A SOM ID automatically is converted to an internal ID representation by the SOM_CheckID macro or by the first invocation of any of the ID manipulation macros. Because the representation of an ID changes, SOM IDs are of a special data type, *somId*.

SOM Debugging

The SOM run-time library provides a means of generating character output by invoking macros and functions that call a replaceable SOM procedure called SOMOutCharRoutine. These macros and functions are available to assist you with debugging an application. Output generated by the debug macros can be conditionally suppressed or produced based on the setting of four global variables. The following table summarizes the SOM debugging macros and the global variables that affect them:

Macro	Control Variables
SOM_Assert	SOM_AssertLevel
SOM_Expect	SOM_WarnLevel
SOM_TestC	SOM_WarnLevel

SOM_WarnMsg	SOM_WarnLevel

In addition to the debug macros, SOM provides a function, somPrintf, that unconditionally generates character output. The interface to somPrintf is identical to the C-library printf routine.

Note: For more detail on debugging SOM, see Debugging Workplace Shell Applications.

SOM Error Handling

The SOM run-time library also provides a way to handle SOM errors by invoking macros (SOM_ERROR and SOM_TEST) that call a replaceable SOM procedure called SomError. SomError produces a message, an error code and, if appropriate, can end the process where the error occurred. SOM errors are classified by severity, which is indicated in the low-order digit of the SOM error code. There are three SOM error classes, or severity levels, as shown in the following table:

Severity Level	Description
SOM_Fatal	Abnormal and irrecoverable.
SOM_Ignore	Normal and informational only.
SOM_Warn	Abnormal but not irrecoverable.

Obtaining SOM Object Information

A SOM class implementer or client program can easily determine the class of an object by invoking the SOM_GetClass macro. This macro returns a pointer to the class object. All SOM class objects support methods, for example, somGetInstanceSize, that return information about objects they create. Therefore, by determining the class of an object and invoking class object methods, more can be learned about the original object.

Replaceable SOM Functions

The SOM run-time environment uses SOM functions that perform memory management, DLL management, character output, and error handling. These functions are replaceable, which means that you can override them by supplying your own version of the default SOM functions. Replaceable SOM functions are summarized in the followig table:

Category	Functions
Character Output	SOMOutCharRoutine
DLL Management	SOMClassInitFuncName
	SOMDeleteModule
	SOMLoadModule
Error Handling	SOMError

Memory Management	SOMCalloc
	SOMFree
	SOMMalloc
	SOMRealloc

Using Workplace Shell and SOM

This section describes how to:

- Program with objects
- Use a release order list
- Process IDL files
- Implement a simple class
- Create and manipulate SOM objects
- Override methods
- Define metaclasses
- Build SOM class libraries
- Convert CSC files to IDL files.

Programming with Objects

An example might make object-oriented programming concepts clearer. A stack is a common programming construct, permitting data to be stored and retrieved in a Last-In, First-Out (LIFO) manner, that is, the last data element placed on the stack is the first element that is retrieved from the stack.

The data structure for the stack describes the stack, a place to store the data put on the stack and a variable to keep track of the location of the top of the stack. Given the definition of the data structure, multiple instances of the stack can be declared within a program.

There are two basic operations that can be performed on a stack: pushing data onto the stack and popping data off from the stack. It also is beneficial to dynamically create a stack. Functions to perform these activities must be defined.

The following sample code shows the definition of a stack data structure and functions and the implementation for the Push function:

```
/* Define the stack */
struct stackType
{
  void *stackArray[STACK_SIZE];
  int stackTop;
};
typedef struct stackType Stack;

/* Define the stack's functions */
Stack *Create();                /* Create a new stack                  */
void Push(Stack *thisStack,     /* Push an element onto the stack      */
          void *nextElement);
void *Pop(Stack *thisStack);    /* Pop an element off from the stack   */

/* The definition of the Push function is provided as an example.  */
/* The rest of the functions would be defined in a similar manner. */
void Push(Stack *thisStack, void *nextElement)
{
```

```
  thisStack stackArray[thisStack stackTop] = nextElement;
  thisStack stackTop++;
}
```

A client program might use this stack to create a stack of words needing interpretation, as in the following sample code:

```
main()
{
  Stack *WordStack;

  char *Subject = "Emily";
  char *Verb    = "eats";
  char *Object  = "ice cream";
  char *NextWord;

  WordStack = Create();
  Push(WordStack, Object);
  Push(WordStack, Verb);
  Push(WordStack, Subject);
     .
     .
     .

  while (NextWord = Pop(WordStack))
  {
    printf("%s\n", NextWord);
     .
     .
     .
  }
}
```

The stack is an example of a class. The stack contains two data elements, *stackArray* and *stackTop* , and supports three methods: Create, Push, and Pop. *WordStack* is an object of class Stack; it also can be called an *instance* of a stack.

Methods must know the specific object on which they are to operate, which is called the *target object* or, sometimes, the *receiving object* . Notice that each method (except Create) takes as its first parameter a pointer to the target object. This is because a program might have many objects of a given class, and each is a potential target for the class methods.

Using a Release Order List

The release order list is specified using the *releaseorder* modifier in the implementation section of the IDL file, as shown in the following sample code. In this example, the release order list groups related methods and data accordingly, that is, by instance variable and the functions that operate on that instance variable.

```
#ifndef  example_idl
#define  example_idl
#include <somobj.idl>

interface Example : SOMObject

{
  attribute string         szVar1;
  attribute unsigned long  ulVar2;

  void Show_All_Vars();

  #ifdef __SOMIDL__
  implementation
  {
    releaseorder:  Show_All_Vars,
                   _get_szVar1,
                   _set_szVar1,
                   _get_ulVar2,
                   _set_ulVar2;

    //# Class modifiers
    callstyle    = oidl;    // For compatibility with SOM1
    majorversion = 1;
```

```
    minorversion = 2;
  };
 #endif /* __SOMIDL__ */
};
#endif  /* Example_idl */
```

If a third attribute variable is added to the class definition file, compatibility with clients of the Example class defined in the previous sample code can be maintained by modifying the release order, as shown in the following sample code:

```
#ifndef  example_idl
#define  example_idl
#include <somobj.idl>

interface Example : SOMObject

{
  attribute string         szVar1;
  attribute char           chVar3;   /* New data added */
  attribute unsigned long  ulVar2;

  void Show_All_Vars();

  #ifdef __SOMIDL__
  implementation
  {
   /* Add new methods at the end of the release order */
   /* list to maintain compatibility with clients     */
    releaseorder:  Show_All_Vars,
                   _get_szVar1,
                   _set_szVar1,
                   _get_ulVar2,
                   _set_ulVar2,
                   _get_chVar3,
                   _set_chVar3;

    //# Class modifiers

    callstyle = oidl;    // For compatibility with SOM1
    majorversion = 1;
    minorversion = 2;
  };
#endif /* __SOMIDL__ */
};
#endif  /* example_idl */
```

In the previous figure, the new attribute *chVar3* is added between the two existing attributes. The new method names, _get_chVar3 and _set_chVar3, are added to the end of the release order list.

Without a release order list, the data and methods in the first version of the Example class are processed in the order in which they occur in the file, as shown in in the following figure:

```
_get_szVar1, _set_szVar1, _get_ulVar2, _set_ulVar2, Show_All_Vars
```

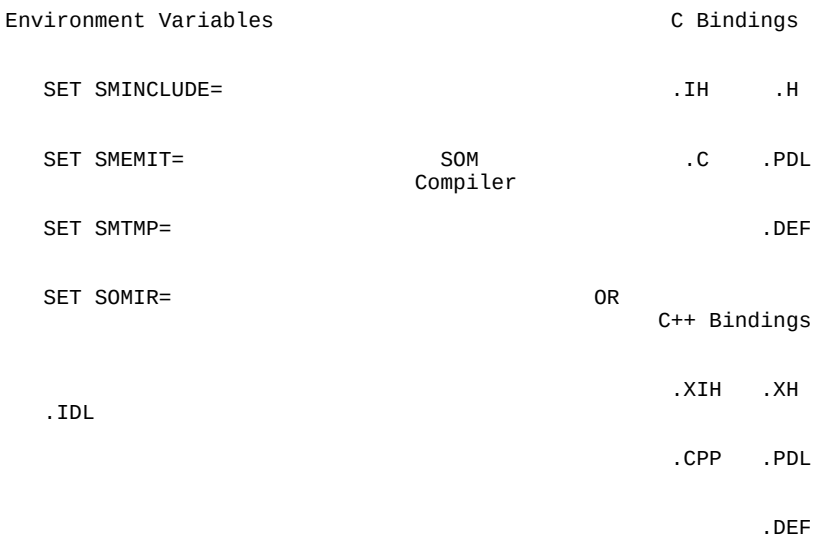
Without a release order list, the methods in the second version of the Example class are processed in the order in which they occur in the file, as shown in in the following figure:

```
_get_szVar1, _set_szVar1, _get_chVar3, _set_Var3, _get_ulVar2, _set_ulVar2, Show_All_Vars
```

The method information maintained by SOM in the object data structures built for the first version of the Example class does not match that built for the second version of the Example class. The second version is not compatible with clients of the first version. Specifying a release order list, and adding new data and methods at the end of the list, ensures client compatibility.

Processing IDL Files

The following figure shows how an IDL file is processed by the SOM compiler:



The SOM compiler processes the IDL file for a SOM class and generates a set of language binding files. The file name of a SOM C-language binding file corresponds to that of the IDL file processed by the SOM compiler, unless the *filestem* modifier was specified in the IDL file. Each SOM C-language file has a different extension. For example, the SOM compiler processes EXAMPLE.IDL and generates EXAMPLE.C, EXAMPLE.H, and so forth. The language binding files generated by the SOM compiler are described in the following table:

C Extension	C++ Extension	Description
.C	.CPP	Template for C-language source program for the class implementation.
.DEF	.DEF	Instructions to the Linker about how to build a class library.
.H	.XH	Public Include file for all C-language programs that need to access the SOM class.
.IH	.XIH	Implementation header containing most of the automatically generated implementation details about the class.
.PDL	.PDL	Language-neutral form and subset of the SOM IDL file with private-implementation detail removed. This file should be "published" (or exported, made available) to users of the class.

Some of these files contain the public interface for the class; others contain the private interface. Some files are used to implement the class and its subclasses, and some are used by client programs that create and manipulate object instances of the class.

The IH and C files are the C-language source files for the class implementation. The IH file is included automatically in the C file. The IDL file is specified in the Include section of the class definition files for subclasses of the class. The H file must be included in client programs to create and manipulate object instances of the class. Separate public and private versions of the .H and .IDL files can be generated by the SOM tools.

A set of environment variables, as shown in the following table, control the SOM-compiler processing.

Variable	Description
SMEMIT	Specifies which output files are to be generated.

```
 SMINCLUDE       Specifies the location of the class
                 definition files (IDL).

 SMTMP           Specifies the directory to be used for
                 intermediate files.

 SOMIR           Specifies the names of interface repository
                 files.
```

SMTMP is optional and defaults to the root directory of the current drive. If the files specified in the Include section of the class definition file are enclosed in double quotation marks, SMINCLUDE is optional and defaults to the root directory of the current drive. If the files specified in the Include section of the class definition file are enclosed in angled brackets (< >), SMINCLUDE is required for the SOM-compiler processing. SMEMIT is used to indicate which bindings files are generated.

Implementing a Simple Class

The language binding files generated by the SOM compiler include a template for the C-language source program for the class implementation. This program template contains stub procedures for all new and override methods specified in the class definition file. You must supply the code implementation for the stub-method procedures. The sample codes illustrated in the following figures show the stages in this process:

```
/*
 * A class defining a set of objects (dogs)
*/
#ifndef dog1_idl
#define dog1_idl
#include <somobj.idl>

interface Dog : SOMObject
{
  void Bark();

  // Have the dog bark

  #ifdef __SOMIDL__
  implementation
  {
    releaseorder:  Bark;

    //# Class modifiers

    callstyle = oidl;    // For compatibility with SOM1
    majorversion = 1;
    minorversion = 2;
  };
  #endif /* __SOMIDL__ */
};
#endif  /* dog1_idl */
```

In the previous sample code, the Dog class is defined. According to this class definition file, the Dog class is derived from the SOMObject class. The behavior of the SOMObject class is defined in the file, SOMOBJ.IDL, which was generated by the SOM compiler when the SOMObject class was implemented. Object instances of the Dog class inherit the behavior of its parent class SOMObject by specifying the SOMOBJ.IDL file in the Include section of the Dog class definition file. This means that all methods that can act on instances of SOMObject can act on the Dog class objects.

In the real world, dogs have many characteristics and behaviors. A comprehensive definition of the Dog class would include methods that relate to these characteristics and behaviors. In this example, the only behavior that is defined is barking. The behavior, "dogs can bark" corresponds to the prototype of the "bark" method specified in the Methods section of the class definition file.

The SOM compiler processes the DOG.IDL file as shown in the previous sample code and generates the DOG.C C-language source-program template shown in the following figure. Notice that DOG.IH, another file generated by the SOM compiler, is included automatically in this program template. This file contains the data structures, macros, and functions for accessing data and methods for object instances of the Dog class. This file also provides stubs for all methods prototyped in the Methods section of the DOG.IDL file.

```
/*
*  This file was generated by the SOM compiler and Emitter framework.
*  Generated using:  SOM Emitter emitctm: 2.38
*/
```

```
#ifndef SOM_Module_dog1_Source
#define SOM_Module_dog1_Source
#endif

#define Dog_Class_Source

#include "dog1.ih"

/*
 * Have the dog bark
*/

SOM_Scope void  SOMLINK Bark(Dog *somSelf)
{
    /* DogData *somThis = DogGetData(somSelf); */
    DogMethodDebug("Dog","Bark");
}
```

The general form for a method stub is shown in the following figure:

```
SOM_Scope void SOMLINK <methodname>(<classname> *somSelf)
{
  <classname>Data *somThis = <classname>GetData(somSelf);
  <classname>MethodDebug("<classname>", "<methodname>");
}
```

SOM_Scope and SOMLINK are C macros used internally with the C-binding files.

Using somSelf

somSelf is a pointer to an object instance of the somSelf class <classname>. Because the same method can be invoked on different objects but implemented differently (polymorphism), a pointer to the object being operated on is required as a parameter in the method invocation. The first parameter in the method invocation always is *somSelf*. This means that if the method prototype in DOG.IDL for the "bark" method is specified as shown in the following figure, the SOM compiler generates the C-language source-program template file as shown in the next figure:

```
void bark(long lCount);
```

```
SOM_Scope void SOMLINK bark(Dog *somSelf, long lCount)
{
  DogData *somThis = DogGetData(somSelf);
  DogMethodDebug("Dog", "Bark");
}
```

Using somThis

The pointer *somThis* points to data for an object instance of the class <classname>. <classname>Data is a class data structure automatically generated by the SOM compiler and placed in the IH file.

Tracing Code

<classname>GetData and <classname>MethodDebug are class macros automatically generated by the SOM compiler and placed in the IH file. <classname>GetData gets the data for the object instance, *somSelf*, of the class.

Note: Methods that access the object's data use the <classname>GetData macro to establish addressability. This macro must be one of the first executable lines of code in each method, and the value it returns should be assigned to a local variable named *somThis*. The SOM compiler automatically generates the code that accomplishes this in each method stub in a C file.

<classname>MethodDebug provides method-tracing capabilities. This custom macro is generated as part of the method stubs produced in the C program template. It takes two arguments: a class name and a method name. If SOM_TraceLevel contains the value 1 or 2, the custom macro produces a message each time a method is entered. Setting SOM_TraceLevel to 2 also causes the methods supplied as part of the SOM run time to generate method trace output. To suppress the generation of method tracing code, place lines similar to the ones shown in the following figure in your C file after the #include statement for <classname>.IH:

```
#undef  <classname>MethodDebug
#define <classname>MethodDebug(c,m) SOM_NoTrace(c,m)
```

Completing the Class Implementation

To complete the class-implementation process, you modify the C-language source-program template as shown in the following sample code. You must supply the code for each of the stubbed method procedures. In the example, "dog barking" is implemented as "printing the sound a dog makes," or, printing "Unknown dog noise." Because the C-library printf routine is used to implement this function, STDIO.H also must be included in the source program.

```
/*
 *  This file was generated by the SOM compiler and Emitter framework.
 *  Generated using: SOM Emitter emitctm: 2.38
 */

#ifndef SOM_Module_dog3_Source
#define SOM_Module_dog3_Source
#endif

#define Dog_Class_Source

#include "dog3.ih"

/* This line is added because of the printf statement below */
#include <stdio.h>

/*
 * Have the dog bark
 */

SOM_Scope void  SOMLINK Bark(Dog *somSelf)
{
   /* DogData *somThis = DogGetData(somSelf); */
   DogMethodDebug("Dog","Bark");
   printf("Unknown dog noise\n");     /* The dog now speaks... */
}
```

Replacing SOM Functions

The SOM run-time environment uses SOM functions that perform memory management, DLL management, character output, and error handling. These functions are replaceable, which means that you can override them by supplying your own version of the default SOM functions.

The following sample code shows how a user-defined function can be substituted for one of the replaceable SOM functions:

```
#define INCL_DOS
#define INCL_WINWORKPLACE

#include <os2.h>
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <string.h>
#include <som.h>

static FILE   *ErrorFile;     /* File to contain debugging error information */

/*
 * This is the replacement for the default SOMOutCharRoutine.
 * It writes the debugging information to a file on the hard drive.
 */

#pragma linkage(myCharacterOutputRoutine, system)

int  SOMLINK  myCharacterOutputRoutine (char chOut)
{
    fputc ((int) chOut, ErrorFile);
    fflush(ErrorFile);
    return 1;   /* Indicate success */
}

/*
 *   Enable myCharacterOutputRoutine by opening the output error
 *   file and then changing the output character routine.
 */

ErrorFile = fopen ("c:\\error.txt", "a" ;

fprintf (ErrorFile, "\nDebug information from my WPS Object\n");
fprintf (ErrorFile,   "-----------------------------------\n");
setbuf (ErrorFile, NULL);

SOM_TraceLevel = 2;   /* Request maximum debugging information */
SOM_WarnLevel  = 2;

/* All output goes to my routine after the next statement */

SOMOutCharRoutine = myCharacterOutputRoutine;
```

Invoking Methods and Accessing Object Data

The basic rules for invoking methods and accessing object data can be summarized as follows:

- Object data can be referred to by preceding the name of the data element with an underscore (_) character. This is valid only in a method of the class, and then only for the object being operated on.
- Methods can be invoked by preceding the method name with an underscore character. Method parameters always include a pointer to the object being operated on. Invocation of instance methods requires a pointer to an instance object. Invocation of class methods requires a pointer to a class object.

These rules can be illustrated by introducing three new methods for the Dog class that relate to more dog characteristics and getting and setting behaviors. Because dogs can be characterized by their breed, a user should be able to get and set the breed for a dog. It also would be desirable to display the characteristics of a dog. In the following sample code, the action of getting and setting a dog's breed corresponds to the prototype for the getBreed and setBreed methods in the DOG.IDL file.

```
/*
 *   A class defining a set of objects (dogs)
 */

#ifndef dog4_idl
#define dog4_idl

#include <somobj.idl>

interface Dog : SOMObject
```

```
{
  attribute string breed;

  // The breed for the dog

  void display();

  // Display the characteristics for this dog.
  // Show the breed and make the dog speak.

  void bark();

  // Have the dog bark
  #ifdef __SOMIDL__
  implementation

  {
    releaseorder:  _get_breed, _set_breed, display, bark;

    //# Class modifiers

    callstyle = oidl;    // For compatibility with SOM1
    majorversion = 1;
    minorversion = 2;
  };
  #endif /* __SOMIDL__ */
};
#endif  /* dog4_idl */
```

The action of displaying a dog's characteristics corresponds to the prototype for the display method. This example also introduces the use of an attribute. Each dog has its own copy of the attribute associated with it. The breed of a dog is stored in a string called "breed". By default, SOM provides a _get_breed method to return the contents of the breed and a _set_breed method to set the contents of breed. If no set method was used, breed should be defined as "readonly attribute" string breed.

From the new DOG.IDL file, the SOM compiler generates a new C-language source-program template, as shown in the following sample code. This template is used to complete the implementation of the new Dog class.

```
/*
 *  This file was generated by the SOM compiler and Emitter Framework.
 *  Generated using:  SOM Emitter emitctm: 2.38
 */

#ifndef SOM_Module_dog_Source
#define SOM_Module_dog_Source
#endif

#define Dog_Class_Source

#include "dog.ih"
#include <stdio.h>   /* Needed because of printf */

/*
 * Display the characteristics for this dog.
 * Show the breed and make the dog speak.
 */

SOM_Scope void  SOMLINK display(Dog *somSelf)
{
  DogData *somThis = DogGetData(somSelf);
  DogMethodDebug("Dog","display");

  /* Note that there are two underscores before */
  /* the get_breed method                       */
  printf("\nMy breed is %s.\n", __get_breed(somSelf));
  printf("I say:\n");

  /* Prefix method name with either an underscore */
  /* ( _ ) or somThis                  */
  _bark(somSelf);
}

/*
 * Have the dog bark
 */

SOM_Scope void  SOMLINK bark(Dog *somSelf)
{
  DogData *somThis = DogGetData(somSelf);
  DogMethodDebug("Dog","bark");
```

```
  printf("Unknown dog noise\n");   /* Speak! */
}
```

The first thing that is noticeable in this sample code is that the attribute declarations are not present. They have been placed in the class-implementation file (DOG.IH) by the SOM compiler. As before, DOG.IH contains all the definitions of the Dog class data, macros, and functions.

The _set_breed and _get_breed methods, also in the DOG.IH file, operate on the string "breed" and refer to it by prefixing the string name with *somThis* .

The display method requires the dog's characteristics, that is, its breed and its bark, to be displayed. To do this, the display method calls both the _get_breed and bark methods, which are referenced by prefixing the method name with the underscore (_) character.

Finally, as in the previous implementation of the Dog class, the Include file, STDIO.H, is added to the source program to resolve the reference to the C-library routine, printf, which is used to display the dog's characteristics. The SOM function, somPrintf, can be used in place of printf; no additional Include file is required in this case. In fact, it is recommended that SOM class implementations use the SOM functions, where appropriate, instead of the C-library functions. SOM functions offer flexibility and replaceability.

Creating and Manipulating SOM Objects

So far, the examples have shown how to define and implement SOM classes but have only mentioned SOM client programs. SOM client programs are applications that create and manipulate SOM objects. A SOM client program can be a Presentation Manager program or a simple C or C++ program.

A SOM client of the new Dog class can, for simplicity, create an instance of the Dog class, define its breed, and display its characteristics. This simple SOM client is shown in the following sample code:

```
/*
 *  DogMain.C - A client C program to manipulate a Dog object
 */

#define INCL_NOPMAPI     /* Makes compile go faster */
                         /* for non-PM programs     */
#define INCL_DOSERRORS
#define INCL_DOS

#include <os2.h>
/* Include declarations for the Dog class */
#include "dog.h"

int main(int argc, char *argv[])
{
  /* Define a dog object called "Zack" */
  Dog  *Zack;

  /* Create an instance of the object */
  Zack = DogNew();

  /* Set the breed of the dog */
  __set_breed(Zack, "Yorkshire Terrier");

  /* Display information about it */
  _display(Zack);

  /* Delete the object */
  _somFree(Zack);

   return NO_ERROR;
}
```

To create and manipulate SOM objects, a client program must have access to the object's public methods. In the same way that class data, methods, and functions are available to class implementers through the .IH file associated with the class implementation, class data methods, and functions are available to client programs through the .H file associated with the class implementation. In the above example, DOG.H is included to resolve references to the dog object's public methods.

The variable *Zack* is defined as a pointer to an instance (object) of the Dog class. In general, a pointer to an instance of a class is declared, as shown in the following figure:

```
<classname> *object;
```

The DogNew Dog class macro then is used to create an instance of the Dog class and return the pointer in the variable *Zack*. DogNew is defined in DOG.H and is a method inherited from the parent (SOMObject) of the Dog class and tailored for the Dog class. The DogNew macro expands to invoke somNew. somNew invokes DogNewClass, which creates the Dog class object, if it has not yet been created. The Dog class object must be created before its instances can be created. If instances of the Dog class are created through some mechanism other than DogNew, DogNewClass must be invoked in the client program.

Because the _set_breed and display methods are public and are defined in DOG.H, the client program can invoke them in the same manner as the class implementation by prefixing the method name with the underscore (_) character. In the client program, the pointer to the dog object (Zack) is the first parameter for these methods. The _set_breed method is called to set Zack's breed as "Yorkshire Terrier". The display method invokes the _get_breed and bark methods and prints the dog's breed and bark.

Finally, somFree releases resources allocated when an object is created by somNew. As previously mentioned, somNew is invoked by the DogNew macro. somFree, like somNew, is a method inherited from the parent, SOMObject, of the Dog class. somFree must be called if somNew is used to create an object.

Compiling a Program

To run the DOGMAIN.C client program, it needs to be compiled and linked with the SOM library and the Dog class implemented in DOG.C. DOG.C is a file generated by the SOM compiler after processing DOG.IDL, and subsequently modified by the class implementor.

The program can be compiled using the IBM CSet/2 compiler. The command line illustrated in the following command line shows how to compile the program assuming that the LIB and INCLUDE environment variables contain the Toolkit directories:

```
icc -Fe dog dogmain.c dog.c somtk.lib
```

These environment variables are generally set appropriately after the Toolkit is installed. If the environment variables do not contain the correct libraries, the steps sequence shown in the following command lines are required:

```
/* Assuming X:\TOOLKIT is the drive and directory */
/* where the toolkit and SOM are installed        */
set sombase=X:\TOOLKIT

set lib=%sombase%\LIB;%lib%
icc -I. -I%sombase%\INCLUDE dog domain.c dog.c somtk.lib
```

The output from the client program is shown in the following figure:

```
My breed is Yorkshire Terrier
I say
Unknown Dog Noise
```

Overriding Methods: Inheritance and Polymorphism

In the Dog class example, the Dog class is derived from SOMObject. Object instances of the Dog class inherit SOMObject behavior, that is, all SOMObject methods can operate on instances of the Dog class. In addition, the Dog class example defined methods not defined for SOMObject. The Dog class is a subclass of SOMObject.

LittleDogs and BigDogs, subclasses of the Dog class, can be defined. These subclasses inherit the behavior of their parent class, Dog class, as well as the behavior of their parent's parent class, SOMObject. If the Dog class had been derived from other classes that were derived from SOMObject, the new subclasses would also inherit the behavior of these ancestor classes. In addition to adding new methods to those inherited from ancestor classes, subclasses can modify or override any inherited methods.

The inheritance relationship between the new subclasses, LittleDog and BigDog and their ancestors, Dog and SOMObject, is shown in the following figure:

```
                                    `       LittleDogs

      SOMObject  `           Dogs
                                    `       BigDogs
```

LittleDogs and BigDogs can be differentiated by the sound, or bark, they make. For this example, instead of making an "Unknown dog noise," little dogs "Yap" and big dogs "WOOF." This means that the LittleDog and BigDog classes implement the bark method inherited from the Dog class differently. In the class definition files for the LittleDog class, this is indicated as a method override, as shown in the following sample code:

```
/*
 *  Define a subclass of Dog for LittleDogs
 */

#ifndef ldog_idl
#define ldog_idl

#include <dog.idl>

interface LittleDog : Dog
{
  #ifdef __SOMIDL__
  implementation

  {
    //# Class modifiers

    callstyle = oidl;    // For compatibility with SOM1
    majorversion = 1;
    minorversion = 2;

    bark: override;
    // Override the bark method
  };
  #endif /* __SOMIDL__ */
};
#endif  /* ldog_idl */
```

In the class definition files for the BigDog class, this is also indicated as a method override, as shown in the following sample code:

```
/*
 *  Define a subclass of the class Dog of BigDogs
 */

#ifndef bdog_idl
#define bdog_idl

#include <dog.idl>

interface BigDog : Dog
{
  #ifdef __SOMIDL__
  implementation

  {
    //# Class modifiers

    callstyle = oidl;    // For compatibility with SOM1
    majorversion = 1;
    minorversion = 2;

    bark: override;
    // Override the bark method
  };
  #endif /* __SOMIDL__ */
};
#endif  /* bdog_idl */
```

Because the parent of the LittleDog and BigDog classes is the Dog class, DOG.IDL must be included in the class definition files so that methods of the Dog class are inherited and can be referenced.

The implementations of the LittleDog and BigDog classes are similar to the implementation of the Dog class. In the implementation of the LittleDog class, as shown in the following figure, the Include file, LDOG.IH, contains the definitions for LittleDog class data, macros, and functions. The implementation of the bark method reflects the yapping of little dogs.

```
/*
 *  This file was generated by the SOM compiler and Emitter Framework.
 *  Generated using:  SOM Emitter emitctm: 2.38
 */

#ifndef SOM_Module_ldog_Source
#define SOM_Module_ldog_Source
#endif

#define LittleDog_Class_Source

#include "ldog.ih"

/*
 * Override the bark method
 */

SOM_Scope void  SOMLINK bark(LittleDog *somSelf)

{
  /* LittleDogData *somThis = LittleDogGetData(somSelf); */
  LittleDogMethodDebug("LittleDog","bark");

  /*  LittleDog_parent_Dog_bark(somSelf);  don't call the parent method */
  somPrintf("yap yap\n");   /* Do small yaps instead */
  somPrintf("yap yap\n");
}
```

In the implementation of the BigDog class, as shown in the following sample code, the Include file, BDOG.IH, contains the definitions for BigDog class data, macros, and functions. The implementation of the bark method reflects the woofing of big dogs.

```
/*
 *  This file was generated by the SOM compiler and Emitter Framework.
 *  Generated using:  SOM Emitter emitctm: 2.38
 */

#ifndef SOM_Module_bdog_Source
#define SOM_Module_bdog_Source
#endif

#define BigDog_Class_Source

#include "bdog.ih"

/*
 * Override the bark method
 */

SOM_Scope void  SOMLINK bark(BigDog *somSelf)
{
  int  i  = 0;   /* Loop index */

  /* BigDogData *somThis = BigDogGetData(somSelf); */
  BigDogMethodDebug("BigDog","bark");

  /* BigDog_parent_Dog_bark(somSelf);  don't call the parent method */
  for (i=0; i<4; i++)
  {
    somPrintf("WOOF WOOF\n");   /* Do woof woof instead */
  }   /* endfor */
}
```

Accessing Public Methods

For a client program of the Dog, BigDog, and LittleDog classes to access public methods for these classes, the client program must include

the public class Header (H) files for the respective classes. In the following sample code, the client of these classes includes DOG.H, LDOG.H, and BDOG.H files.

```
/*
 *  DogsMain.C - A client C program to manipulate Dog, LittleDog,
 *               and BigDog objects
 */

#define INCL_NOPMAPI        /* Makes compilation go faster */
                            /* for non-PM programs         */
#define INCL_DOSERRORS
#define INCL_DOS

#include <os2.h>
/* Include declarations for the Dog class       */
#include "dog.h"
/* Include declarations for the LittleDog class */
#include "ldog.h"
/* Include declarations for the BigDog class    */
#include "bdog.h"

int main(VOID)
{
  Dog        *Pokey;   /* Define a Dog       */
  LittleDog  *Zack;    /* Define a LittleDog */
  BigDog     *Pepper;  /* Define a BigDog    */

  /* Instantiate some objects */
  Pokey  = DogNew();
  Zack   = LittleDogNew();
  Pepper = BigDogNew();

  /* Set their breeds */
   __set_breed(Pokey, "Basset Hound");
   __set_breed(Zack, "Yorkshire Terrier");
   __set_breed(Pepper, "Rottweiler");

  /* Display the dog's characteristics */
   _display(Pokey);
   _display(Zack);
   _display(Pepper);

  /* Free the dogs  */
   _somFree(Pokey);
   _somFree(Zack);
   _somFree(Pepper);

   return NO_ERROR;
}
```

The next figure shows the output from this client program.

```
My breed is Basset Hound
I say
Unknown Dog Noise

My breed is Yorkshire Terrier
I say
yap yap
yap yap

My breed is Rottweiler
I say
WOOF WOOF
WOOF WOOF
WOOF WOOF
WOOF WOOF
```

Defining Metaclasses

In the implementation of the Dog, BigDog, and LittleDog classes, a metaclass is not specified in the Metaclass section of the class definition files. This means that the metaclass of each of these classes is the metaclass of their parent class. The metaclass of the BigDog and LittleDog classes is the metaclass of its parent class, Dog. The metaclass of the Dog class is the metaclass of its parent class, SOMObject. The metaclass of the SOMObject class is SOMClass. SOMClass, then, supplies the constructors for the Dog, BigDog, and LittleDog classes, as well as for SOMObject.

If a metaclass is specified in the Metaclass section of the IDL files, a new metaclass is defined for the classes. The new metaclass provides new constructors for the classes. As an example, a new metaclass M_Dog is defined for the Dog class and added to DOG.IDL. The following sample code shows the new DOG.IDL file, including the changes made to the original DOG.IDL file.

```
/*
 *  Define a metaclass of the Dog class.
 *  By convention, metaclasses are the class name prefixed with M_
 */

#ifndef dogmeta_idl
#define dogmeta_idl

#include <somcls.idl>
#include <somobj.idl>

interface Dog;   /* Declared later in this file */
interface M_Dog : SOMClass
{
  Dog   CreateADog();

  // Create an instance of the Dog class and return a pointer to it
  #ifdef __SOMIDL__
  implementation

  {
    releaseorder:  CreateADog;

    //# Class modifiers

    callstyle = oidl;    // For compatibility with SOM1
    majorversion = 1;
    minorversion = 2;
  };
  #endif   /* __SOMIDL__ */
};

/*
 *   A class defining a set of objects (dogs) with metaclass M_Dog
 */

interface Dog : SOMObject
{
  attribute string breed;

  // The breed for the dog
  void display();

  // Display characteristics for this dog.
  // Show the breed and make the dog speak.
  void bark();

  // Have the dog bark
  #ifdef __SOMIDL__
  implementation

  {
    releaseorder:  _get_breed, _set_breed, display, bark;

    //# Class modifiers

    callstyle = oidl;    // For compatibility with SOM1
    majorversion = 1;
    minorversion = 2;
    metaclass = M_Dog;   // Explicitly specify a metaclass
  };
  #endif   /* __SOMIDL__ */
};
#endif   /* dogmeta.idl */
```

Defining Methods

Because SOMClass is the root class for all metaclasses, M_Dog is derived from SOMClass. This is reflected in the Parent and Include sections in the class definition file for the M_Dog class. The only method that M_Dog defines is CreateADog, that is, the constructor that creates an instance of the dog class, as shown in the following sample code:

```
/*
 * This file was generated by the SOM compiler and Emitter Framework.
 * Generated using:  SOM Emitter emitctm: 2.38
 */

#ifndef SOM_Module_dogmeta_Source
#define SOM_Module_dogmeta_Source
#endif

#define M_Dog_Class_Source
#define Dog_Class_Source

#include "dogmeta.ih"

/*
 * Create an instance of the Dog class and return a pointer to it
 */

SOM_Scope Dog*  SOMLINK CreateADog(M_Dog *somSelf)
{
  Dog   *aNewDog;
  aNewDog = _somNew(somSelf);

  /* M_DogData *somThis = M_DogGetData(somSelf); */
  M_DogMethodDebug("M_Dog","CreateADog");

  /* Return statement to be customized */
  return(aNewDog);
}

/*
 * Display characteristics for this dog.
 * Show the breed and make the dog speak.
 */

SOM_Scope void  SOMLINK display(Dog *somSelf)
{
  DogData *somThis = DogGetData(somSelf);
  DogMethodDebug("Dog","display");

  /* Note there are two underscores before the get_breed below */
  printf("\nMy breed is %s.\n", __get_breed(somSelf));
  printf("I say:\n");
  /* Prefix method name with either an underscore or somThis  */
  _bark(somSelf);
}

/*
 * Have the dog bark
 */

SOM_Scope void  SOMLINK bark(Dog *somSelf)
{
  DogData *somThis = DogGetData(somSelf);
  DogMethodDebug("Dog","bark");
  printf("Unknown dog noise\n");   /* Speak! */
}
```

Implementing Metaclasses

The implementation of the M_Dog class is similar to the implementation of the Dog class. However, DOGMETA.C is also a client of the Dog

class; CreateADog returns a pointer to an instance of the Dog class.

The final implementation of the new Dog class is identical to the previous example. However, clients of the Dog class instantiate instances of the Dog class differently, as shown in the following sample code:

```
/*
 *  DogMMain.C - Another client C program to manipulate a Dog object
 */

/* Makes compile go faster for non-PM programs */
#define INCL_NOPMAPI
#define INCL_DOSERRORS
#define INCL_DOS

#include <os2.h>
/* Include declarations for Dog and M_Dog */
#include "dogmeta.h"

int main(VOID)
{
  Dog  *Zack;

  /*
   * Create the Dog class and save a pointer to it.
   */

  M_Dog *DogClassObj;
  DogClassObj = DogNewClass(Dog_MajorVersion, Dog_MinorVersion);
  Zack = _CreateADog(DogClassObj);
  __set_breed(Zack, "Yorkshire Terrier");
  _display(Zack);
  _somFree(Zack);
  return NO_ERROR;
}
```

The constructor method, that is, the method that creates instances of an object, for the Dog class is now CreateADog. In the previous examples, the DogNew macro was used to instantiate Dog objects. Because the DogNew macro invokes DogNewClass to create the Dog class object, the previous client programs did not have to invoke DogNewClass directly. Because the implementation of CreateADog does not call DogNewClass, the new client program is required to do so.

The parameters for DogNewClass are defined by the class implementer in the DOGMETA.IDL file. MajorVersion and MinorVersion are attributes defined in the attribute section of the IDL file and used by DogNewClass to determine compatibility with versions of the class library.

Building SOM Class Libraries

Two basic rules explain the process of building SOM class libraries:

- IDL files for a class' metaclass, parent class, or ancestor class are required for the SOM compiler to process a class definition file. These files establish the relationships between classes.
- H files are required for C-language compiler processing and building of the class library. These files resolve references to methods defined in other files.

With these rules in mind, the basic process for building a SOM class library can be summarized as follows:

1. Create IDL files and do SOM compilation for all parent and ancestor classes and their related classes.
2. Create IDL files for a class and its metaclass (if any).
3. Compile the metaclass IDL file using SOM.
4. Compile the class IDL file using SOM.
5. Compile C implementation files, in any order, using the C compiler.

Converting CSC Files to IDL Files

Object and class definitions written in the OS/2 operating system and prior to OS/2 operating system Version 3, were defined in IBM object interface definition language (OIDL) and placed in a class definition file, also called a CSC file. This file was then used as input to the SOM compiler to generate language binding files.

SOM was enhanced to conform with the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA**) standards. This required changing the syntax of the class definition language from IBM OIDL to CORBA's interface definition language (IDL). The bindings created by the SOM compiler were also changed to make them compatible with the CORBA standards.

Class definitions written in OIDL and residing in CSC files should be converted to IDL before being used with the SOM compiler. This conversion allows the use of multiple inheritance, exception handling, type checking, and automatic descriptor support. In addition, the programs generated from IDL class descriptions are significantly smaller and run faster than programs generated from OIDL class definitions.

CTOI Program

The conversion from a .CSC file to an .IDL file is handled by the CTOI program. For example, to convert the file MYDOG.CSC to an IDL file, make sure all the CSC, SC, and PSC files referenced by MYDOG.CSC are accessible, and then invoke the program as shown in the following command line:

```
ctoi mydog.csc
```

Note: Converting a Workplace Shell application from a CSC file to an IDL file requires the SC files from the previous OS/2 Toolkit installed on your system (in the SC directory), and all the ones that you might have created. The IDL subdirectory from the OS/2 Toolkit contains the IDL files for the public Workplace Shell classes and objects.

In this example, the CTOI program produces a file called MYDOG.IDL containing the definitions in IDL format. In many cases, you should be able to compile, install and run your application without further modification. However, the following situations will require manual intervention:

- Any classes that use IDL reserved words as functions or variable names need to be changed. For example, "string", "context", "interface".
- The SOM compiler no longer generates .SC, .PSC, or .PH files. Installation and make files that refer to .CSC, .SC, .PSC, or .PH files need to be changed.
- The data declaration changes. Public and private instance variables are converted to attributes. However, structures and arrays require manual changes, as shown in the following figure:

```
OIDL                            IDL

data:
  struct info infoBuffer;       interface: ...   {
                                   typedef struct info info;
                                   ...
                                implementation {
                                   info infoBuffer;
                                   ...
                                   };

data:
  char achMyArray[20];          interface: ...   {
                                   typedef char myarraytype[20];
                                   ...
                                implementation {
                                   attribute myarraytype achMyArray;
                                   ...
                                   };

data:
  unsigned char chFlag;         attribute octet chFlag;

data:
  char    *argv[];              char   **argv;
```

- IDL does not permit structures to be passed by value. Instead, methods must pass a pointer to a structure. Methods can return a structure, and if a method does so, it should have the modifier *struct* attached to it, as shown in the following figure:

```
AStructure resolveMyIdentity(in string pszIdent), struct;
```

- The OIDL private qualifier has been removed. Private instance data and methods must now be enclosed in a special #ifdef directive, as shown in the following figure:

OIDL	**IDL**
`int x, private;`	`#ifdef __PRIVATE__`
	`  int x;`
`VOID mymethod(), private;`	`  VOID mymethod();`
	`#endif`

 To include private data and methods in the processing of an IDL file, the SOM compiler should be invoked with either the -P option or the -D_PRIVATE_ option, as shown in the following command lines:

```
sc -p yourname.idl

sc -d__PRIVATE__ yourname.idl
```

 Note that there are two underscore (__) characters before and after the word PRIVATE.

- Passthru statement processing has changed. Passthru statements should only be used to pass #include directives to the language binding files. In addition:
 - Information can no longer be directed to the implementation (.C) file
 - Most information directed to implementation Header (.IH) files should be moved to the implementation (.C) file
- Forward references are required in IDL. Thus, classes and objects must be declared before being referenced. The following figure shows an example to forward reference the M_Cat class:

```
interface M_Cat;
```

Because IDL is strongly typed (unlike OIDL), data structures and constants should be declared before referencing them, in order to eliminate unnecessary warning messages during compilation. This is required if an interface repository (.IR) file is to be created from the IDL file.

Workplace Shell Distributed SOM

Distributed SOM (DSOM) provides a framework that allows application programs to access objects across address spaces. That is, application programs can access objects in other processes, even on different machines. Both the location and implementation of an object are hidden from the client which accesses the object (using method calls) in the same manner regardless of its location.

About Workplace Shell Distributed SOM

DSOM currently supports two types of distribution:

- Distribution among processes on the same machine, which is referred to as Workstation DSOM
- Distribution among a network of machines, which is referred to as Workgroup DSOM.

Only Workstation DSOM is provided with the OS/2 operating system.

DSOM is initially disabled in the OS/2 operating system. There are two reasons for this:

- Performance
- DSOM can be easily activated as required. Not all users will require DSOM to be active. Therefore, they should not be penalized with the additional system overhead.

DSOM packaged with the OS/2 operating system consists of two parts:

- DSOM daemon which is also referred to as SOMDD
- Workplace Shell DSOM server (WPDS).

The DSOM daemon (SOMDD) has a very important role and must be running at all times in order to communicate between processes.

The Workplace Shell DSOM server is a generic server whose server class is wpdServer and is adequate for many environments. The Workplace Shell DSOM server behaves as a simple server. In that, it simply receives requests and executes them synchronously. The server creates its server object from the derived class SOMDServer and will load any class libraries it needs on demand. For more complex environments where the default server does not provide the needed functions, the *SOMObjects Developer Toolkit User's Guide* provides details for writing your own DSOM server.

Starting DSOM

The Workplace Shell, when started, will default to the wpdServer class and load WPDSERV.DLL. If you create a new server class, you can tell the Workplace Shell to use your server by setting WPDSERVERALIAS to YourServer in the CONFIG.SYS file and use *regimpl* to change the server name from wpdServer to YourServer.

There are a number of ways to start the DSOM daemon and Workplace Shell DSOM server. To start both SOMDD and WPDS at the time the OS/2 operating system is started, set WPDSERVERSTART to ON in the CONFIG.SYS file and reboot. Both the daemon and the server are hidden to the user when started.

If during development you require to see the output of SOMDD, then set the environment variable SOMDD.DISPLAY to ON in the CONFIG.SYS file and reboot. When using the Workplace Shell DSOM server you should always allow the Workplace Shell to start the DSOM daemon and never start the daemon from a command line. In this way, the Workplace Shell is aware of its existence.

In the case, when a client DSOM application is run and only the DSOM daemon is active, then the implementation repository, using the implementation ID passed to it to locate the server definition, starts an executable, WPDSACTV.EXE, which in-turn starts the Workplace Shell DSOM server. Once the server is active, then the client program will continue to process as normal.

The OS/2 operating system supplies a utility called WPDSINIT.EXE which provides the functions to start, stop and query the current status of the DSOM daemon and the Workplace Shell DSOM server from the command line:

Function Name	**Description**
WPDSINIT /S	To start both the daemon and the server.
WPDSINIT /K	To stop both the server and the daemon.
WPDSINIT /Q	To query the current status of both the daemon and the server.

The DSOM daemon or the Workplace Shell DSOM server can be manipulated independently by specifying either *somdd* or *server* as the second parameter to WPDSINIT.

The last method available for manipulating both the daemon and the server is programmatically through four new functions:

- WinIsSOMDDReady
- WinRestartSOMDD
- WinIsWPDServerReady
- WinRestartWPDServer

Performance Using DSOM

In a distributed environment, a designer needs to keep in mind the possible performance impacts, because IPC communications is always slower than procedure calls within the same process. Writing a normal Workplace Shell application that runs within the Workplace Shell's process always perform better than an application in another process using DSOM.

Initializing a Client Workplace Shell Program

A DSOM application will not execute if the underlying DSOM support is not active. Therefore a good client application might query and save the current state of the DSOM daemon and Workplace Shell DSOM server and start them, if required before issuing any DSOM calls.

A client application must declare and initialize the DSOM run-time before attempting to create or access a remote object. The SOMD_Init procedure initializes all of the DSOM run-time. A client application must also merge the SOM class manager with the Workplace Shell class manager and initialize all application classes used by the program. For each class, the corresponding <classname>NewClass call should be made. When creating a Workplace Shell DSOM application, each class referenced in the hierarchy must be initialized.

Exiting from a Client Workplace Shell Program

At the end of a client program, the SOMD_Uninit procedure must be called to free DSOM run-time objects, and to release system resources such as semaphores, shared memory segments, and so on. No special action is required for the Workplace Shell.

Using Workplace Shell Distributed SOM

The sample creates and opens a new folder on the Desktop, and then inserts a number of different class objects into that folder, one of which is opened after it is created. It also creates and deletes an abstract object on the Desktop before terminating.

Note: The sample uses the four new functions that are available for manipulating the daemon and the server:

- WinIsSOMDDReady
- WinRestartSOMDD
- WinIsWPDServerReady
- WinRestartWPDServer

This sample Workplace Shell DSOM program demonstrates the minimum Workplace Shell code required to initialize a simple Workplace Shell DSOM application. To ensure that it will always run, regardless of the current state of the DSOM server, the sample queries and saves the state of both the DSOM daemon and the Workplace Shell DSOM server. The state is saved, so that it can be reset before the program terminates.

Once the server is active, a local DSOM environment is created by calling SOM_CreateLocalEnvironment and then register itself with the DSOM daemon by calling SOMD_Init. Before any Workplace Shell method calls can be made, the Workplace Shell class manager must be merged with the SOM's class manager, and then Workplace Shell classes on which method calls will be made must be initialized. For instance, if you are writing an application that creates an abstract object on the Desktop and then terminates, both WPObjectNewClass and WPAbstractNewClass would need to be called because WPAbstract is a descendant of WPObject.

Next a server must be located to execute the methods on, so _somdFindServerByName is called to find the Workplace Shell DSOM server wpdServer. To query the Desktop object, first the class object for WPFolder must be acquired by calling _somdGetClassObj. With this pointer, finally the first Workplace Shell method _wpclsQueryFolder is called to acquire a pointer to the Desktop folder. Once the pointer to the Desktop folder is obtained, objects are created using _wpclsNew. As soon as the folder object on the Desktop is created, it uses that object pointer as the target for all the other objects created. A Details view of the folder is created using _wpSetup and the new created color palette is opened with a call to _wpOpen. The abstract object created on the Desktop is finally destroyed with a call to _wpDelete.

Before terminating, the DSOM environment should be cleaned up by calling the reverse of initialization:

- _somdReleaseObject is called to free the server object
- SOMD_Uninit is called to free the DSOM resources
- SOM_DestroyLocalEnvironment is called to free the local DSOM environment.

The sample also resets the state of the DSOM daemon and Workplace Shell DSOM server before terminating. In other words, if the server was active when the sample application was run, then it will remain active when it terminates. If on the other hand it was not running, then it will be stopped before the program terminates. This may not be the desired result in most applications. If you are running a number of DSOM client applications back to back, then you will not want ythe application stopping the server each time. In most cases, you should leave the

server active after it is started.

Source Code for WPDS

The following example shows the source modules for the sample Workplace Shell DSOM program:

```
/*
 * Purpose..: To demonstrate how to call Workplace Shell
 * methods using DSOM.
 *
 * Workplace Shell methods called:
 *        _wpclsQueryFolder()
 *        _wpclsNew()
 *        _wpSetup()
 *        _wpOpen()
 *        _wpDelete()
 */

#pragma info(nogen)

#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>
#define  INCL_DOS
#define  INCL_WINWORKPLACE
#define  INCL_PM
#include <os2.h>
#include <somd.h>
#include <somtcnst.h>
#include <wpobject.h>
#include <wpdesk.h>
#include <wpabs.h>
#include <wpfolder.h>
#include <wptrans.h>
#include <wpmet.h>
#include <wpclrpal.h>
#include <wpdataf.h>
#include <wpclsmgr.h>

enum {STOP, START};

WPClassManager *vWPClassManagerObject;

BOOL       checkForError(Environment *, char *, ...);
SOMClass * getClassObject(SOMClass *, Environment *, string);

int main(int argc, char *argv[])
{
  Environment *ev;
  SOMClass   *Server;
  SOMObject  *objDesktop;
  SOMObject  *objWPAbs;
  SOMObject  *objWPFolder;
  SOMObject  *object;
  M_WPFolder *classFolder;
  APIRET     apiRtnCd;
  BOOL       fDaemonUp;
  BOOL       fServerUp;

  /*
   * Check if the DSOM daemon is active.
   * If not, it needs to be started before continuing.
   */
  fDaemonUp = WinIsSOMDDReady();
  if (!fDaemonUp)
  {
    apiRtnCd = WinRestartSOMDD(START);
    if (apiRtnCd)
    {
      somPrintf("Unable to start the DSOM dameon, rc=%ld\n", apiRtnCd);
      exit(apiRtnCd);
    }
  }

  /*
```

```
 * Check if the Workplace Shell DSOM server is up and running.
 * If not, it needs to be started before continuing.
 */
fServerUp = WinIsWPDServerReady();
if (!fServerUp)
{
  apiRtnCd = WinRestartWPDServer(START);
  if (apiRtnCd)
  {
    somPrintf("Unable to start the DSOM server, rc=%ld\n", apiRtnCd ;
    exit(apiRtnCd);
  }
}

/*
 * Create the local DSOM environment
 */
ev = SOM_CreateLocalEnvironment();
SOMD_Init(ev);

/*
 * Merge the Workplace Shell class manager with the
 * SOM class manager
 */
vWPClassManagerObject = WPClassManagerNew();
_somMergeInto(SOMClassMgrObject, vWPClassManagerObject);

/*
 * Initialize all the Workplace Shell classes used by the program
 */
WPObjectNewClass( 1, 1 );
WPTransientNewClass( 1, 1 );
WPAbstractNewClass( 1, 1 );
WPFileSystemNewClass( 1, 1 );
WPDataFileNewClass( 1, 1 );
WPFolderNewClass( 1, 1 );
WPMetNewClass( 1, 1 );
WPDesktopNewClass( 1, 1 );

/*
 * Find the Workplace Shell DSOM server by its name
 */
Server = _somdFindServerByName(SOMD_ObjectMgr, ev, "wpdServer");
checkForError(ev, "Could not find server 'wpdServer'\n");

/*
 * Get a pointer to the WPFolder's class object
 */
classFolder = getClassObject(Server, ev, "WPFolder");

/*
 * Get the object pointer of the Desktop folder
 */
objDesktop = _wpclsQueryFolder(classFolder, "<WP_DESKTOP>", TRUE);
if (objDesktop)
{
  /* Create an abstract object on the Desktop */
  somPrintf("Creating a WPAbstract object on Desktop: ");
  objWPAbs = _wpclsNew(getClassObject(Server, ev, "WPAbstract"),
                       "WPAbstract Object",   /* Title of the object */
                       "",                    /* Setup string        */
                       objDesktop,            /* Target folder       */
                       TRUE);                 /* Never dormant       */
  somPrintf("%s\n", objWPAbs ? "Succeeded" : "\aFailed");

  /*
   * Create a WPFolder object on the Desktop
   */
  somPrintf("Creating a opened WPFolder object on Desktop: ");
  objWPFolder = _wpclsNew(classFolder,                /* Object's class name  */
                          "WPFolder Object",          /* Title of the object  */
                          "OPEN=ICON;ALWAYSSORT=YES", /* Setup string         */
                          objDesktop,                 /* Target folder        */
                          TRUE );                     /* Never dormant        */
  somPrintf("%s\n", objWPFolder ? "Succeeded" : "\aFailed");

  /*
   * Create a WPDataFile object in the new folder on the Desktop
   */
  somPrintf("Creating a WPDataFile object in folder: ");
  object = _wpclsNew(getClassObject(Server, ev, "WPDataFile"),
                     "WPDataFile Object",   /* Title of the object */
                     "",                    /* Setup string        */
```

```
                    objWPFolder,           /* Target folder       */
                    TRUE);                 /* Never dormant       */
  somPrintf("%s\n", object ? "Succeeded" : "\aFailed");

  /*
   * Create a WPTransient object in the folder
   */
  somPrintf("Creating a WPTransient object in folder: ");
  object = _wpclsNew(getClassObject(Server, ev, "WPTransient"),
                    "WPTransient Object",   /* Title of the object */
                    "",                     /* setup string        */
                    objWPFolder,            /* Target folder       */
                    TRUE);                  /* Never dormant       */
  somPrintf( "%s\n", object ? "Succeeded" : "\aFailed" );

  /*
   * Create a metafile object in the folder
   */
  somPrintf("Creating a WPMet object in folder: ");
  object = _wpclsNew(getClassObject(Server, ev, "WPMet"),
                    "WPMet Object",   /* Title of the object */
                    "",               /* setup string        */
                    objWPFolder,      /* Target folder       */
                    TRUE);            /* Never dormant       */
  somPrintf("%s\n", object ? "Succeeded" : "\aFailed");

  /*
   * Open a Detail view of the folder
   */
  _wpSetup(objWPFolder, "OPEN=DETAILS");

  /*
   * Create a color palette in the folder and open palette
   */
  somPrintf("Creating a WPColorPalette object in folder: ");
  object = _wpclsNew(getClassObject(Server, ev, "WPColorPalette") ,
                    "WPColorPalette Object",   /* Title of the object */
                    "",                        /* Setup string        */
                    objWPFolder,               /* Target folder       */
                    TRUE);                     /* Never dormant       */

  somPrintf("%s\n", object ? "Succeeded" : "\aFailed");
  _wpOpen(object, NULLHANDLE, OPEN_DEFAULT, 0L);
  somPrintf("Deleting the WPAbstract object on the Desktop.\n");
  _wpDelete(objWPAbs, 0);
}
else
{
  somPrintf("\aUnable to acquire the Desktop object pointer\n");
}

/*
 * Clean up the DSOM environment
 */
_somdReleaseObject(SOMD_ObjectMgr, ev, Server);
checkForError(ev, "_somdReleaseObject failed\n");
SOMD_Uninit(ev);
SOM_DestroyLocalEnvironment(ev);

/*
 * If the Workplace Shell DSOM server was started
 * then, lets stop it before terminating
 */
if (!fServerUp)
{
  apiRtnCd = WinRestartWPDServer(STOP);
  if (apiRtnCd)
    somPrintf("Stopping the DSOM server was unsucessful, rc=%ld\n", apiRtnCd);
}

if (!fDaemonUp)
{
  ULONG count = 0;

  /*
   * Loop until the server thread ends or the program times out
   */
  while(WinIsWPDServerReady())
  {
    HEV hev;

    /*
     * First create a private, reset, event semaphore.
```

```
       * Wait for 1 second, then try again for a maximum
       * of 30 seconds
       */
      DosCreateEventSem((PSZ) NULL, &hev, 0, FALSE);
      DosWaitEventSem(hev, 1000);
      if (count++ > 30)
        break;
    }
    apiRtnCd = WinRestartSOMDD(STOP);
    if (apiRtnCd)
      somPrintf("Stopping the DSOM daemon was unsucessful, rc=%ld\n", apiRtnCd);
  }
  return 0;
} /* End of main() */

/* getClassObject */
SOMClass *getClassObject(SOMClass *Server, Environment *ev, string sClassName)
{
  SOMClass *ClassObj = NULL;
  ClassObj = _somdGetClassObj(Server, ev, sClassName);
  if (checkForError(ev, NULL) || ClassObj == NULL)
  {
    somPrintf("\aCould not find class '%s'\n", sClassName);
    ClassObj = NULL;
  }
  return ClassObj;
}

/* Check for errors */
BOOL checkForError(Environment *ev, char *pchFmt, ...)
{
  va_list  vargs;
  char *exId;
  APIRET apiRtnCd = FALSE;

  switch(ev->_major)
  {
    case SYSTEM_EXCEPTION:
    {
      StExcep *params;
      exId   = somExceptionId(ev);
      params = somExceptionValue(ev);
      somPrintf("\a** Error Occurred **\n");
      somPrintf("** exception string: %s\n", exId);
      somPrintf("** minor error code: %u\n", params->minor);
      somPrintf("** completion code: %s\n",
                (params->completed == YES ? "YES" :
                 params->completed == NO ? "NO": "MAYBE" ));
      somExceptionFree(ev);

      if (pchFmt)
      {
        char achBuf[1024];
        va_start(vargs, pchFmt);
        vsprintf(achBuf, pchFmt, vargs);
        somPrintf(achBuf);
        va_end(vargs);
        SOMD_Uninit(ev);
        SOM_DestroyLocalEnvironment(ev);
        exit(1);
      }
      apiRtnCd = TRUE;
      break;
    }

    case USER_EXCEPTION:
    {
      if (pchFmt)
      {
        char achBuf[1024];
        va_start(vargs, pchFmt);
        vsprintf(achBuf, pchFmt, vargs);
        somPrintf(achBuf);
        va_end(vargs);
      }
      apiRtnCd = TRUE;
      break;
    }

    default:
    case NO_EXCEPTION:
    {
      apiRtnCd = FALSE;
```

```
      break;
    }
  }
  return(apiRtnCd);
}
#pragma info(nouse)
```

Resource Definition File for WPDS

The following example shows the resource definition file:

```
NAME WPDS WINDOWCOMPAT
DESCRIPTION 'WPDS: Sample Workplace Shell DSOM program.'
PROTMODE
CODE LOADONCALL MOVEABLE DISCARDABLE
DATA MULTIPLE NONSHARED LOADONCALL
```

Make File for WPDS

The following example shows the make file:

```
app     = wpds
TOOLKIT = c:\toolkit
IBMCPP  = c:\ibmcpp

CC      = icc
LINKER  = link386
LIBLIST = os2386 + somtk + dde4mbs
INCLUDE = $(TOOLKIT)\h;$(IBMCPP)\include;.;$(INCLUDE)
LIB     = $(TOOLKIT)\lib;$(IBMCPP)\lib;$(LIB)
CFLAGS  = -D_OS2 -c -Q+ -Ss+ -Gd- -Gm+ -Gs+ -Sm -Kb -Ms -Ge+
LFLAGS  = /exepack /packd /packc /align:4 /stack:360000 /pm:vio /noi /nol /map

#
# Set the local environment here
#
!if [set INCLUDE=$(INCLUDE);.] || \
    [set LIB=$(LIB)] || \
    [set PATH=$(TOOLKIT)\bin;$(IBMCPP)\bin;%path%]
!endif

.SUFFIXES:  .obj .c .h

.c.obj:     $(CC) $(CFLAGS) $<

all:        $(app).exe

$(app).obj: $(app).c
            $(CC) $(CFLAGS) $(app).c

$(app).exe: $(app).obj
            $(LINKER) $(LFLAGS) $(app).obj,$(app).exe,$(app).map,$(LIBLIST),$(app);
            mapsym $(app).map
```

Workplace Shell Processes and Threads

The Workplace Shell's current implementation exploits the multithreading capabilities of the OS/2 operating system. This chapter describes how the Workplace Shell is initialized, the relationship between its various threads, and ramifications on Workplace Shell method invocation.

About Workplace Shell Processes and Threads

The *Workplace process* is the one under which all the Workplace Shell classes are loaded and initialized. Therefore, objects representing Workplace Shell classes and their subclasses must run on this process. The Workplace process is actually launched from the *Shell process* , which is the process indicated in the SET PROTSHELL= statement in the CONFIG.SYS file. Once the Shell process is running, it starts the Workplace process. It is the Shell process that is responsible for restarting the Workplace process in the event that it is ended as a result of a trap. The PROTSHELL= statement in the CONFIG.SYS file indicates which process is to be launched as the Shell process. The SET RUNWORKPLACE= statement in the CONFIG.SYS file indicates which process is to be the Workplace process. In the default configuration, both the PROTSHELL and RUNWORKPLACE environment variables are set to PMSHELL.EXE. PMSHELL.EXE is designed to distinguish between being started as the Shell process versus being started as the Workplace process. The following diagram illustrates the initialization sequence:

```
  Shell Process                        Workplace Process

PMSHELL.EXE Session
                                              PMSHELL.EXE Session

                                              If (Workplace_Process)
                                                  Start Workplace Shell


Create Window List                              SOM Is Initialized
Start Starter Thread                            Classes Are Loaded
                                                Desktop Is Opened
        Starter Thread                          Restart List Is Processed
        {                                       Etc...
          While (TRUE)
          {
            If (Workplace Process Is Not Running)
              Launch PMSHELL.EXE;
          }
        }
```

The Shell Process

While the Shell process can be any application (xxx.EXE), PMSHELL.EXE is the default. This section describes the behavior of PMSHELL.EXE specifically. The important features of the Shell process are as follows:

- The Starter thread is always running on the main Shell process.
- If the Starter thread detects that the Workplace process is not started or has died, it will automatically restart it.

The threads in this section are initiated from the Shell process and are closely tied to the functioning of the Workplace process.

The Starter Thread

The Starter thread runs on the Shell process and is mainly responsible for starting the Workplace process. It will automatically restart the

Workplace process in the event that the:

- Workplace process might trap
- User code running on the Workplace process might trap.

In addition, the Starter thread is responsible for starting all applications as children of the first Shell process instead of children of the Workplace process. Thus, when the Workplace process needs to be restarted, all applications that were run from it continue to function normally, while the Workplace process is restarted.

Since applications that were initiated by a user action in the Workplace Shell are really started from this thread, when such applications are terminated, the Starter thread must pass a corresponding notification back to the Workplace process.

The Shutdown Thread

When a shutdown is initiated, the Shutdown thread is created on the Shell process. The *Shutdown thread* closes all running applications in the system. The Shutdown thread creates a PM message queue. It broadcasts to all applications a message instructing them to save their data, initiates the closing of all open folders, and so on.

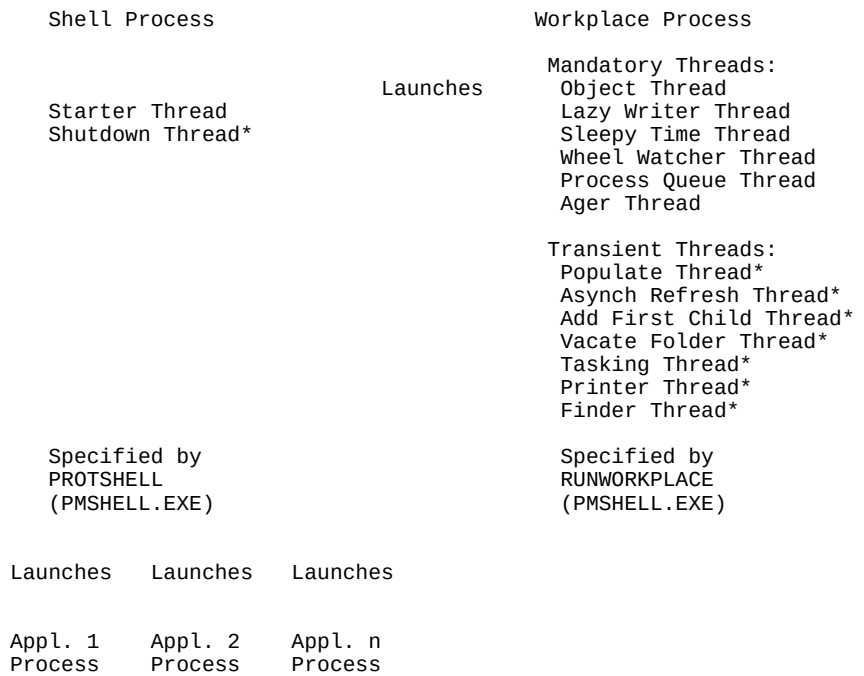
Each child process of the Shell process will send a WM_APPTERMINATENOTIFY to the Shell process when it goes away. Unless a shutdown is in effect, when the Shell process receives such notification for the Workplace process, it will restart the Workplace process.

The Workplace Process

When started, the Workplace process' first thread becomes the User-interface thread. The *User-interface thread* is the one responsible for monitoring the Workplace Shell's PM message queue and dispatching the appropriate actions based on which messages come through. The User-interface thread is the one that most noticeably affects the user's perception of response time. Therefore, it is mandatory that only the minimum amount of processing takes place on that thread. There are a set of other threads that are also started at different times during the lifetime of the Workplace process. There are divided into two major categories:

- Mandatory threads which are always running
- Transient threads which are started and stopped over time.

The following diagram shows the Shell and Workplace process and their threads. The asterisk (*) indicates the transient threads.



```
APP1.EXE    APP2.EXE    APPn.EXE
```

Mandatory Threads

The mandatory threads are as follows:

- Object
- Lazy writer
- Sleepy time
- Wheel watcher
- Process queue
- Ager.

The Object Thread

The *Object thread* represents the Workplace object window. It is not visible. It is used to offload I/O intensive operations from the main user-interface thread. The Object thread has a PM message queue. It utilizes an event semaphore, which it uses to signal the Sleepy time thread when the object thread is idle. The Sleepy time thread uses this event semaphore to throttle itself so that it does not fill the Object thread's message queue with make dormant messages, which would implicitly keep the Object thread's priority higher than the main User-interface thread's priority, and cause visibly noticeable delays in response time.

Some of the notifications received and processed by the Object thread are as follows:

- Receives notification from PMSHAPI when program groups or their contents are added or deleted
- Receives notification from the Starter thread when an application that the Workplace process asked to run gets terminated
- Receives notification that an object needs to be made dormant
- Processes messages for class loading, unloading and replacement which are sent by functions such as WinRegisterObjectClass and WinReplaceObjectClass
- Receives notification that a file system notification needs to be processed in the Workplace Shell
- Processes messages sent by functions such as WinCreateObject, WinDestroyObject, and WinSetObjectData
- Processes some of the shutdown process.

The Lazy Writer Thread

The *Lazy writer thread* is responsible for writing objects' state in its persistent form. A wpSaveDeferred results in an object subsequently being processed by this thread. When the Lazy writer thread stores an object, it will call _wpSaveState. This thread does not have a PM message queue. Synchronization is controlled by event semaphores that are signaled from various other threads. The Lazy writer thread detects periods of file system inactivity, as well as other heuristics, and at such times performs actions such as:

- Storing file system objects' handles to the OS2SYS.INI file

- Storing abstract objects' handles to the OS2.INI file
- Saving all objects on the Dirty list
- Saving the restart list
- Saving any updated window positions to the OS2.INI file
- Updating the PM_Workplace:Locations entry in the OS2.INI file.

The Sleepy Time Thread

An object, in its dormant state, consists of a block of data written in a persistent form, for example, a series of appnames in the INI file, or a file's extended attributes. Conversely, when an object is awake, it is instantiated in memory. To reduce the number of objects that exist in memory, when objects are no longer visually exposed, they are added to a list of *sleeping objects* . The *Sleepy time thread* maintains this list of objects that may need to be made dormant, and progressively ages them until they are either removed from the sleepy object list, or until they need to be made dormant. Incidentally, when an object is locked it cannot be made dormant. This prevents objects that may not be visibly exposed to remain in memory on demand. The Sleepy time thread sends a notification to the Object thread, which will cause the object to be made dormant, when it has been on the sleepy object list for the OBJECTSNOOZETIME period specified in the CONFIG.SYS file.

The Wheel Watcher Thread

The *Wheel watcher thread* is responsible for catching notifications from the file system regarding the **Move**, **Copy**, **Delete**, **Rename**, ... commands. Only the Workplace process in the operating system can receive these notifications. File system notifications are then added to a system queue, which is monitored by another thread on the Workplace process, the Process queue thread. The Wheel watcher thread does not maintain a PM message queue.

The Process Queue Thread

The *Process queue thread* takes file system notifications from a queue and uses heuristics to determine if successive notifications:

- Are redundant
- Cancel each other out
- Are necessary, for example, if the folder containing the object in question is not awake.

If not, a notification is then handed off to a third thread, the Ager thread. The Process queue thread does not maintain a PM message queue.

The Ager Thread

The *Ager thread* is responsible for aging notifications for an appropriate amount of time so that intermediate file system modifications are not unnecessarily processed by the Workplace Shell. For example, removing the possibility of a short-lived, application generated file from appearing and then immediately disappearing from the view of an open folder. When a notification has aged appropriately, the Ager thread notifies the Object thread that a notification is ready. The actual processing of the notification takes place from the Object thread, upon receiving such notification.

Transient Threads

Other threads' lifetimes are brief and may be started and restarted many times. The Transient threads are as follows:

- Populate
- Asynch refresh
- Add first child
- Vacate folder
- Tasking
- Printer
- Finder.

The Populate Thread

The *Populate thread* is created when a folder is opened. It is from this thread that _wpPopulate is invoked. wpPopulate is responsible for finding and instantiating from their persistent state all objects that reside in a particular folder. Because this is very I/O intensive, this is handled asynchronously. The Populate thread is responsible for sending messages to the container instructing it to add new records representing the newly instantiated objects.

The Asynch Refresh Thread

The Asynch refresh thread may be created by wpPopulate when the folder, for which the populate was requested, was already previously populated, and no open views currently exist. The *Asynch refresh thread* simply invokes wpRefresh on a specified folder. This thread creates a PM message queue. It is also initiated when **Refresh now** is selected from an open folder's pop-up menu.

The Add First Child Thread

The *Add first child thread* is initiated when it is requested that a folder, opened in Tree view, be populated. Because in the open Tree view, every visible folder containing subfolders must have a plus (+) sign next to it, each visible folder must be populate one level deep to see if it contains at least one subfolder. Because there can be many such folders, this processing is done on a thread other than the main User-interface thread.

The Add first child thread is also created when a folder is added to or removed from:

- An open Tree view
- One of the subfolders in an open Tree view,

so that its plus sign can be modified appropriately. This thread uses a round robin queue to fairly service folders that need processing. This is so that if several folders are opened in Tree view, visible subfolders will be processed in turn, rather than one view being handled at a time. When this queue is full, _wpAddFirstChild is invoked synchronously. The Add first child thread creates a PM message queue.

The Vacate Folder Thread

The Vacate folder thread is invoked when a folder is closed. The *Vacate folder thread* saves the folder's icon positions, along with other information pertaining to its window position. It then sends a message to the folder's container causing it to invoke WinDestroyWindow on itself.

The Tasking Thread

The *Tasking thread* is invoked to perform user-initiated tasks such as **Copy**, **Move**, **Link**, and **Create another** operations on a separate thread, so that it does not interfere with the main User-interface thread's response time. Therefore, the methods that are associated with these actions can be executed from other threads.

The Printer Thread

The *Printer thread* is created to perform the actual printing of a file (as a result of a _wpPrintObject). It creates a PM message queue.

The Finder Thread

The *Finder thread* is created when a **Find** operation is initiated via the user interface. This thread performs the I/O intensive operation of actually finding objects that match specifications. The Finder thread creates a PM message queue.

Using Workplace Shell Processes and Threads

This section describes ramifications of multithreading on Workplace Shell method invocation and specifies notes for developers of user-defined threads.

Multithreading on Workplace Shell Method Invocation

Workplace Shell application development involves subclassing existing classes, introducing new methods as well as defining behavior for overridden Workplace Shell methods. In writing these method overrides, it is important to keep in mind that they may be invoked from other threads. For example, an override of wpPopulate will be invoked on an object of a WPFolder subclass, when wpOpen or wpViewObject is invoked. As described in "The Populate Thread" section, wpPopulate is sometimes invoked from the Populate thread.

Developing User-Defined Threads

Because Workplace Shell objects are running under the Workplace process, they can be accessed by any thread belonging to that process. And, as such, may have methods invoked upon them from any of these threads. Some Workplace Shell methods require the thread in which they were invoked to have a PM message queue. Therefore, when writing your own threads, in your subclass' implementation, make sure to do the following, even if your thread does not need a message loop:

```
hab = WinInitialize(0);
if (hab)
{
  hmq = WinCreateMsqQueue(hab,0);
  if (hmq)
  {
    WinCancelShutdown(hmq, TRUE);
```

```
        .
        .
        .
  }
}
```

Note:

- Exception handling should be added to any user-defined threads, so that traps originating in that code do not end up hanging the Workplace process.
- Per thread and per process variables are accessible via DosGetInfoBlocks, which yields a TIB structure.
- Threads require a 4KB stack each.
- When calling DosSetPriority, do not use idle priority threads.
- DosExit on a thread does not free resources nor close files; DosExit on a process does.

Workplace Shell Programming Interface

The Desktop (which also is an object), the objects that appear on the Desktop, and the underlying code supporting these objects constitute the OS/2 Workplace Shell, the default OS/2 user interface. Likewise, methods in the Workplace Shell are methods that were implemented for the creation of the Desktop object and its objects. This chapter describes Workplace Shell classes and characteristic behaviors of Workplace Shell objects. Because Workplace Shell objects are built using the IBM System Object Model (SOM), the OS/2 Workplace Shell Programming Interface requires a knowledge of SOM. It assumes a knowledge of object-oriented programming and design concepts, as well as SOM.

About Workplace Shell Programming Interface

To establish a base definition of an object-oriented user interface, CUA guidelines are highlighted. The OS/2 Workplace Shell is an object-oriented user environment based on these guidelines. General object-oriented user interface concepts are then used to explain the underlying design of the Workplace Shell Programming Interface.

CUA Guidelines for an Object-Oriented User Interface

CUA guidelines defined the user interface for *application-oriented* environments. In object-oriented environments, such as the OS/2 operating system, the user performs tasks by manipulating on-screen objects. The user does not start an application to perform a task. Instead, the user can do the following:

- Select an action or task that can be performed on an object. For example, the user selects the **Open** action on an ASCII file object. The operating system then starts an editor to work on the file.
- Select the **Open** action on a spreadsheet file. The operating system then starts a spreadsheet application to work on the file.
- Drag the ASCII file object to a printer object and drop it on the printer object. The file is then sent to the printer.

The concept of applications is transparent to the user. The user interacts with objects rather than with the operating system or with separate applications. The user focuses on the task and not on the tools used to perform the task. The user interacts with objects in the same manner across tasks.

Objects, Classes, Hierarchies, and Inheritance

An *object* is any visual component of the user interface with which the user can work, independent of other items, to perform a task. An object can be represented by one or more graphic images, called *icons* . The user can interact with an object (or its icon) just as the user can interact with objects in the real world.

Views of Objects

The user also can interact with an object by opening a window that displays more information about the object. The content of a window is a *view* of an object. A view is a way of looking at an object's information. Different views display information in different forms, just as information about an object is presented in the real world. An object can have more than one view.

The appearance of a window's contents and the kinds of interaction possible in a window are determined, in part, by the type of view presented in the window. CUA guidelines describe four basic types of views:

- Composed Views

 A *Composed view* of a data object arranges the object's data in an order that conveys the data's meaning. If the data is arranged differently in a Composed view, the object has a different meaning. For example, a graph or chart object typically is displayed in a Composed view because the arrangement of the components determines the meaning of the object as a whole. If the arrangement of the components changes, the meaning of the object changes.

- Contents Views

 A *Contents view* lists the components of an object. The components can be ordered or unordered in the view; the order of the information displayed in a Contents view does not affect the meaning of the object containing the information. CUA guidelines describe two kinds of Contents views:

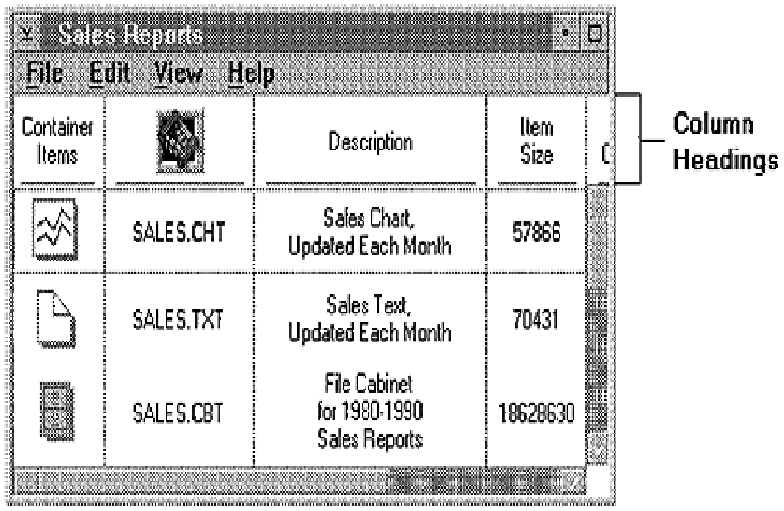
 - Icon view

 An *Icon view* displays each object as an icon. Its purpose is to give the user an easy way to change the position of objects or to otherwise directly manipulate them. An object usually is represented by only one icon. However, for some tasks, the user might find it convenient to represent an object with more than one icon. For example, the user might want a representation of a printer object in more than one place so that the user can have easy access to the printer from anywhere. The user can create an additional icon, known as a *shadow* , to represent the same printer object. The following figure shows an Icon view:



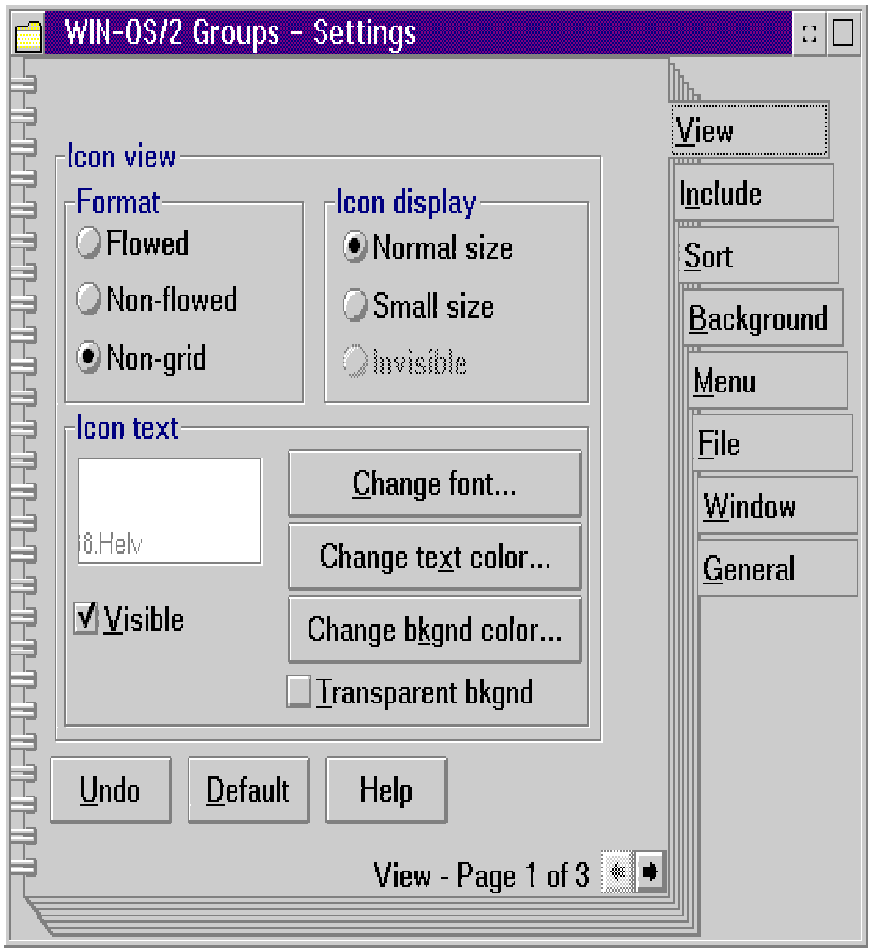
- Details view

 A *Details view* combines small icons with text that provides additional information about objects. The type of information displayed depends on the type of object and the type of tasks the user wants to perform. A Details view gives the user access to some of the object's more frequently used information, without requiring the user to open the object. Small icons are included in a Details view to provide a way for the user to easily recognize objects and to directly manipulate each object. The following figure shows a Details view:



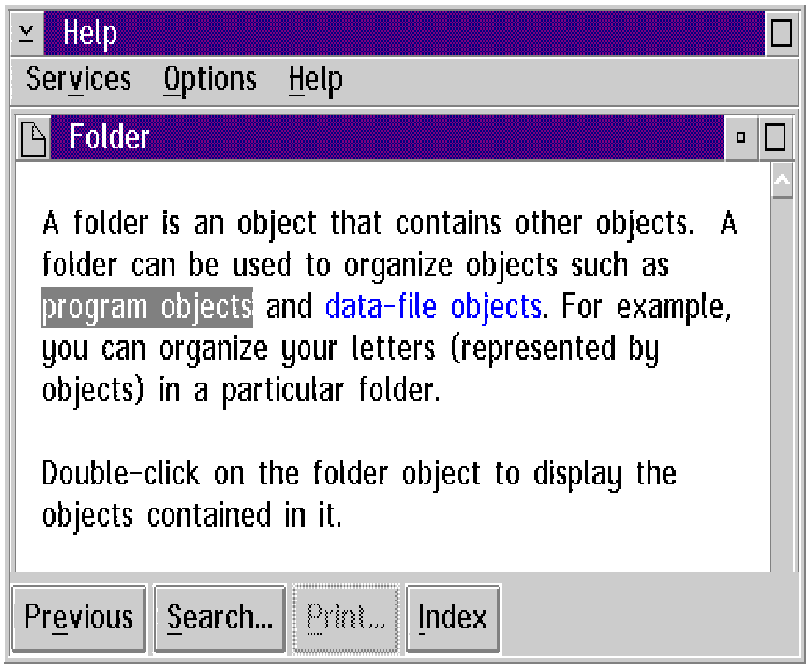
- Settings views

 A *Settings view* displays information about the characteristics, attributes, or properties of an object, and provides a way for the user to change the settings of some characteristics or properties. A Settings view typically is provided for each type of object. The following figure shows a Settings view of an objects:



- Help views

 A *Help view* provides information that can assist the user in working with an object. The type of information displayed in a help view depends on the type of help the user requests. For example, the user can request help for an entire window or for part of a window. The following figure shows a help view assisting the user in working with an object:

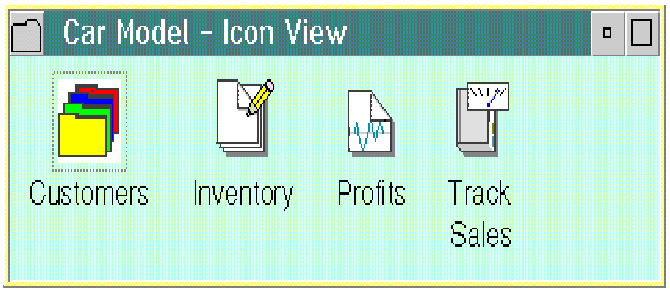


Classes of Objects

Objects can be classified by similarities in characteristics and behaviors. Each class of objects has a primary purpose that distinguishes it from other classes, and all three types of objects can contain other objects. CUA guidelines define three object classes:

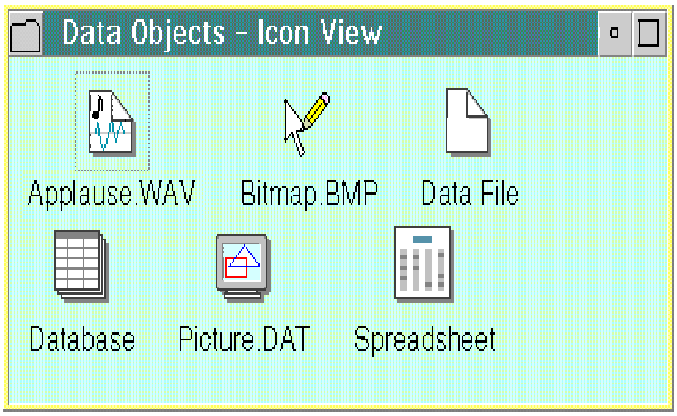
- Container objects

 A *Container object* holds other objects. Its primary purpose is to provide a way for the user to group related objects for easy access and retrieval. The following figure shows a Container object:



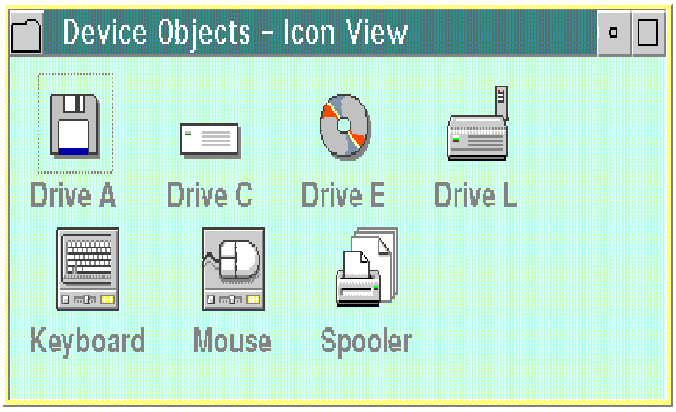
- Data objects

 Data objects convey information, such as text or graphics, audio or video information. The following figure shows a Data object:



- Device objects

 A *Device object* often represents a physical object in the real world. For example, a mouse object can represent the user's pointing device, and a modem object can represent the user's modem. Some device objects represent a logical object in the user's computer system rather than a physical object. For example, a shredder object can represent a logical object that disposes of the user's other objects. Some device objects can contain other objects. For example, a printer object can contain a queue of objects to be printed. The following figure shows a Device object:



Object Relationships

An object class can be defined in terms of another object class. It can be derived from another class, inheriting the same characteristics and behaviors of the other class, yet having characteristics and behaviors of its own. The class that it is derived from is called its *parent class* ; the class itself is referred to as a *subclass* of its parent class.

The inheritance relationship between objects is hierarchical. An object that is lower in the inheritance hierarchy than another object has all of the characteristics of the object or objects above it and can have new characteristics of its own.

Other object relationships also can be hierarchical. For example, objects can be arranged in a containment hierarchy that illustrates which objects can contain which other objects.

Interaction with Objects

In an object-oriented user environment, users interact with objects to perform tasks. The object-action paradigm is a pattern for interaction in which the user first selects an object, and then selects an action. When the user selects an object, the system can then present a list of actions that can be applied to that object. Some actions might require the user to respond to additional choices. Interaction with an object through choices and controls is known as *indirect manipulation* .

At any given time an object has a set of actions that can be performed on it. Different objects have different actions that can be performed on

them. Action choices for an object are displayed in pop-up menus that appear next to an object when the user presses the appropriate key or mouse button. The content of a pop-up menu is based on the object's context, which includes its current state, location, and contents.

Users also can interact with objects by way of a pointing device. This is known as *direct manipulation* . This interaction technique closely resembles the way the user interacts with objects in the real world. For example, the user can *pick up* an object and put it into a folder. This technique is also known as *dragging* an object from one place and *dropping* it at another place.

A Drag and Drop operation often involves a *source object* and a *target object* . A *source object* usually is the object the user is working with, and a *target object* usually is an object to which the user is transferring information. For example, the user drags a file object to a printer object so that the file can be printed; the file is the source object and the printer is the target object.

The result of a Drag and Drop operation can change depending on what the source object is and what the target object is. For example, if the user drags a file object from one folder object and drops it on another, the file is moved to the target folder. However, if the user drops the same file onto a printer, the operating system makes a copy of the file and puts the copy into the printer's queue to be printed. The original file is returned to its original location.

Designing an Object-Oriented User Interface

The following items are key to designing an object-oriented user interface:

- Objects and their relationships
- Visual representations of objects
- Interaction techniques and mechanisms.

Objects and their relationships can be defined by answering the following questions:

- What objects does the user require?
- How are the objects related?
- What properties and behaviors should the objects have?

To illustrate the importance of objects and their relationships, consider the example of the design of a software model of a car dealership. A salesperson needs a car object to represent each car model on his lot. He also needs a customer object to represent each customer that purchases a car. He needs a worksheet object to track sales, profits, inventory, customers, and so on. Finally, he needs container objects to group these objects.

The visual representations of objects must ensure consistency with one another and with the operating system. Visual representations of objects should address the functional aspects of visual representations, such as usability and purpose: Does the visual convey the purpose of the object being represented? Visual representations also should address the aesthetic aspects, such as shape, size, and color: Is the representation visually pleasing?

Users should interact with similar objects in similar ways and in ways that seem natural. Users also should have a choice of interaction mechanisms that suits their tasks, level of skill, and preferred style of interaction. In the car dealership example, a salesperson must be able to place information into a worksheet object in any of several ways. The salesperson can:

- Place a car object on top of the worksheet object, thereby transferring information about the car to the worksheet
- Type the information directly into the worksheet object
- Select portions of information from the car object and copy them to the worksheet object.

Defining Objects for a Software Model

To design a software model that serves the needs of the users, the car dealership example is translated into:

- Who are the users?
- What tasks do they perform?

In this example, the salesperson is the user. The user's tasks can include:

- Determining what a customer wants, needs, and can afford
- Determining what cars are in stock that match the customer's wants, needs, and budget
- Negotiating an agreement using a worksheet

- Getting approval from the sales manager
- Giving the worksheet information to the finance manager.

From this analysis, the objects that should be part of our software model can be identified. These include a car object, car lot object, customer object, customer list object, worksheet object, worksheet list object, salesperson object, sales manager object, and finance manager object.

To simplify this explanation, consider the car object only. Each car object represents a real car for sale in the car lot. A car object contains descriptive information about the corresponding real car, such as its year, make, model, price, factory-installed options, color, and vehicle identification number. Because the primary purpose of a car object is to convey information, the car object is a data object.

Determining Object Relationships and Behaviors

The next step in the design of a software model is to determine how each object interacts with other objects. In the car dealership, information contained in car objects must be transferable to a worksheet object. The user should have the option to drag a car object and drop it onto a worksheet object. The user also should have the option to choose an action to copy the information from a car object to a worksheet object.

Determining the Necessary Views

After identifying and defining the objects, consider what views of the car object will give the user (salesperson) the best access to the objects and the information they contain. Car information consists of a combination of text (model, year, and so forth) and graphic information (a picture of a car) that make up a single, General information view.

Determining the Action Choices

From a salesperson's perspective, most of the information about a car object is fixed, that is, the information is based on a real-world object and cannot be changed unless something changes about the real-world object. For example, it would not make sense to permit a salesperson to change the color of a car object to correspond to the color of the actual car he is trying to sell. Because a salesperson can change little about a car object, the car object has only a few action choices:

- Open as General information view
- Print
- Edit
- Copy to the clipboard
- Find
- Windows.

OS/2 Object-Oriented User Interface: The Workplace Shell

In the OS/2 operating system, the *Desktop* is a collection of objects (icons) and windows associated with those objects. The Desktop (which also is an object), the objects that appear on the Desktop, and the underlying code supporting these objects constitute the OS/2 Workplace Shell, the default OS/2 user interface.

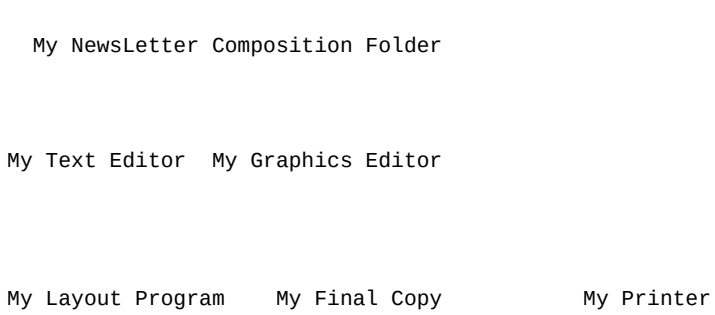
The OS/2 Workplace Shell provides an object-oriented user interface. It provides a seamless environment, where all services are task-oriented and the user is shielded from the complexities of the operating system. The user can perform tasks faster and easier and with a shorter learning curve.

In the OS/2 Workplace Shell, applications from OS/2 operating system are replaced by objects and collections of objects, or *folders* . Users do not have to be aware of where an object is (which drive or network) or what it is called. They can place an object wherever they wish and call

it by any name. They do not have to know about EXEs, DLLs, device drivers, or how to add a printer or use a network. If they want to print a report on the laser printer down the hall, they can simply drag the icon representing the report and drop it onto the icon representing the laser printer that is labeled "Laser printer down the hall".

Users act on all objects in a consistent manner. They can act on program files the same as program references. There is no difference in using programs on a network server or on a hard disk or on a CD drive. There is no difference in setting up or printing to a local or remote printer.

In the OS/2 Workplace Shell, users are not aware of the file system. They do not need to know anything about disk drives or directories. They need to know only about folders and the objects they put into them. They can put applications, files, and so forth, in a folder, as shown in the following figure. They can arrange things, regardless of where they physically reside, to suit themselves and their own needs. And they can label the folders by any name.



Some of the objects that the OS/2 Workplace Shell provides are described in the following table. After installation, some of the objects appear directly on the Desktop. Some of them are contained in folders. Users can re-arrange and re-label objects to suit themselves.

Object	Description
Clock	Sets and views the current date, time, and alarm.
Color palette	Sets and views colors for visual elements of user interface and applies color to a window.
Country	Sets and views international conventions for system elements (country, date, time, numbers).
Font palette	Sets and views fonts for textual elements of user interface and apply fonts to windows.
Keyboard	Sets and views keyboard configuration (timing, mappings, special needs).
Mouse	Sets and views behavior of mouse device (timing, setup, button mappings).
Printer	Sets and views a print destination (a print queue and its associated port).
Scheme palette	Sets and views window color and font attributes.
Shredder	Destroys an object.
Sound	Enables/disables warning beep.
Special needs	Explains implications of special needs mode when activated.
Spooler	Enables/disables spooling. Sets and views spool path.
System	Sets and views behavior of system elements (confirmations, logo, windows).

Using OS/2 Workplace Shell Programming Interface

While object-oriented user interfaces share some concepts with object-oriented programming, user objects might not necessarily correspond to software objects. Object-oriented programming can make the development of an object-oriented user interface easier. However, an object-oriented user interface can be developed with more traditional programming languages and tools.

The OS/2 Workplace Shell is an example of a user interface developed using object-oriented programming, specifically, the IBM System Object Model. In fact, every user object in the OS/2 Workplace Shell is an instance of a Workplace Shell software class object. There is a one to one correspondence between Workplace Shell (user) objects and Workplace Shell (software) classes. See *Workplace Shell Reference* for a complete class hierarchy of Workplace Shell objects.

All Workplace Shell classes are derived from a root Workplace Shell class, WPObject, which is derived from the root SOM class, SOMObject. Workplace Shell classes are defined using the SOM's object interface definition language (OIDL). Workplace Shell class libraries are built using the SOM compiler. Workplace Shell objects are instantiated by the Workplace Shell on behalf of the user through the Workplace Shell class list object, installation programs, or batch files. The same rules that apply to SOM classes apply to Workplace Shell classes, with the exception that applications cannot call Workplace Shell methods. (SOM client applications can call SOM methods.)

Some Workplace Shell classes (WPObject, WPAbstract, WPFileSystem, WPTransient) cannot be instantiated. These classes are provided as base classes which define common characteristics and behaviors for descendant classes. Object characteristics and behaviors are defined as methods for the object's class, as well as methods inherited from ancestor classes.

WPObject is the root class for all Workplace Shell classes. It defines behavior common to all Workplace Shell objects. The immediate descendants of WPObject are storage classes, from which all other Workplace Shell classes are derived. *Storage classes* define methods for storing and retrieving data associated with instances of descendant classes. Storage classes provided with the OS/2 Workplace Shell are shown in the following table:

Class	Storage of Object Instance Data
WPAbstract	Object instance data stored in user profile (OS2.INI).
WPFileSystem	Object instance data stored in files in the file system.
WPTransient	Object instance data not saved.

Objects whose instance data and state is preserved between system shutdown and system startup are called *persistent objects.* Objects whose instance data and state need not be preserved between system shutdown and system startup are called *non-persistent objects* .

Designing Workplace Shell Classes

To design a Workplace Shell class, first identify all the actions to which an object instance can respond. Based on this list, define the methods in the class definition file that correspond to the actions that were identified. To illustrate this process and understand how method requirements for a class are identified, consider the WPObject and WPAbstract classes.

Based on the general description of user objects in a CUA-conforming user environment, objects have the following:

- Properties (for example, an icon representation in the Workplace Shell)
- Views that contain information about the object
- Pop-up context menus that describe actions to which the object can respond
- Mobility (direct manipulation).

These characteristics and behaviors should be reflected in the methods in the class definition file for the WPObject class. The following table shows the mapping of characteristics and behaviors common to all Workplace Shell objects defined for this class:

Method Group	Method Name	Characteristic
Container	wpCnrInsertObject	Object containers
	wpCnrRemoveObject	
	wpCnrSetEmphasis	
	wpRegisterView	
	wpclsInsertMultipleObjects	
Details	wpIsSortAttribAvailable	Object criteria

	wpQueryDetailsData	
	wpQueryFldrDetailsClass	
	wpQueryFldrSort	
	wpQueryFldrSortClass	
	wpSetFldrDetailsClass	
	wpSetFldrSort	
	wpSetFldrSortClass	
	wpSetSortAttribAvailable	
	wpclsQueryDetails	
	wpclsQueryDetailsInfo	
Direct manipulation	wpDragCell	Object mobility
	wpDraggedOverObject	
	wpDragOver	
	wpDrop	
	wpDroppedOnObject	
	wpEndConversation	
	wpFormatDragItem	
	wpRender	
	wpRenderComplete	
Error handling	wpQueryError	Object errors
	wpSetError	
	wpclsQueryError	
	wpclsSetError	
Help	wpDisplayHelp	Object aid
	wpMenuItemHelpSelected	
	wpQueryDefaultHelp	
	wpSetDefaultHelp	
	wpclsQueryDefaultHelp	
Memory allocation	wpAllocMem	Object memory
	wpFreeMem	
Notebook	wpAddObjectGeneralPage	Object properties
	wpAddObjectGeneralPage2	
	wpAddObjectWindowPage	
	wpAddSettingsPages	
	wpCnrRefreshDetails	
	wpInsertSettingsPage	
Pop-Up menu	wpClose	Object user action
	wpCopyObject	
	wpCreateFromTemplate	
	wpCreateShadowObject	

	wpDelete	
	wpFilterPopupMenu	
	wpHide	
	wpInsertPopupMenuItems	
	wpMenuItemSelected	
	wpModifyPopupMenu	
	wpMoveObject	
	wpOpen	
	wpPrintObject	
	wpRestore	
	wpViewObject	
Save/Restore state	wpRestoreData	Object persistence
	wpRestoreLong	
	wpRestoreState	
	wpRestoreString	
	wpSaveData	
	wpSaveDeferred	
	wpSaveImmediate	
	wpSaveLong	
	wpSaveState	
	wpSaveString	
Setup/Cleanup	wpFree	Object initialization and termination
	wpInitData	
	wpIsObjectInitialized	
	wpObjectReady	
	wpScanSetupString	
	wpSetup	
	wpSetupOnce	
	wpUnInitData	
	wpclsInitData	
	wpclsUnInitData	
Usage	wpAddToObjUseList	Object usage
	wpCnrDeleteUseItem	
	wpDeleteFromObjUseList	
	wpFindUseItem	
	wpFindViewItem	
	wpIsLocked	
	wpLockObject	
	wpUnlockObject	
Set/Query	wpModifyStyle	Object

wpQueryButtonAppearance

wpQueryConcurrentView

wpQueryConfirmations

wpQueryContainerFlagPtr

wpQueryCoreRecord

wpQueryDefaultHelp

wpQueryDefaultIconPos

wpQueryDefaultView

wpQueryDetailsData

wpQueryError

wpQueryFolder

wpQueryHandle

wpQueryIcon

wpQueryIconData

wpQueryMinWindow

wpQueryNameClashOptions

wpQueryObjectID

wpQueryScreenGroupID

wpQueryStyle

wpQueryTitle

wpSetButtonAppearance

wpSetConcurrentView

wpSetDefaultHelp

wpSetDefaultIconPos

wpSetDefaultView

wpSetError

wpSetIcon

wpSetIconData

wpSetMinWindow

wpSetObjectID

wpSetStyle

wpSetTitle

wpclsQueryButtonAppearance

wpclsQueryDefaultHelp

wpclsQueryDefaultView

wpclsQueryDetails

wpclsQueryDetailsInfo

wpclsQueryError

wpclsQueryExtendedCriteria

wpclsQueryFolder

wpclsQueryIcon

```
wpclsQueryIconData

wpclsQueryObject

wpclsQuerySettingsPageSize

wpclsQueryStyle

wpclsQueryTitle

wpclsSetError

wpclsSetIcon

wpclsSetIconData

wpclsSetSettingsPageSize
```

Workplace Shell Win Functions

Applications cannot call Workplace Shell objects' methods directly. They are not clients of Workplace Shell objects, in the same sense that applications can be clients of SOM objects. Workplace Shell objects are derived from the WPObject class, which, in turn, is derived from the SOMObject class. They share all the features of SOM objects but only the Workplace Shell can directly manipulate them. Because there are times when applications might need to effect changes to the Desktop and objects on the Desktop, the Workplace Shell provides functions that permit you to proceed these changes. This chapter describes the Workplace Shell Win Functions.

About Workplace Shell Win Functions

Outside the Workplace Shell environment, objects are DLLs that consist of data and code that operates on that data when objects are instantiated in the Workplace Shell run-time environment. Workplace Shell objects have no meaning outside the Workplace Shell environment. Workplace Shell classes take form when the class is registered with the Workplace Shell and the class is instantiated. The Workplace Shell and SOM provide the underlying code (predefined Workplace Shell objects' methods) that supports an object.

The Workplace Shell calls the appropriate object's methods when the user interacts with the object. In this sense, the Workplace Shell is the client of all Workplace Shell objects. The Workplace Shell manipulates the object (its code) on behalf of its users.

On the other hand, applications cannot call Workplace Shell objects' methods directly. They are not clients of Workplace Shell objects, in the same sense that applications can be clients of SOM objects. Workplace Shell objects are derived from the WPObject class, which, in turn, is derived from the SOMObject class. They share all the features of SOM objects-encapsulation, inheritance, and polymorphism-but only the Workplace Shell can directly manipulate them.

Because there are times when applications might need to effect changes to the Desktop and objects on the Desktop, the Workplace Shell provides functions that permit you to proceed these changes.

Object Instance Functions

Applications can create and destroy object instances of Workplace Shell classes by calling WinCreateObject and WinDestroyObject. To create an object, an application must specify the object's:

- Class name
- Title
- Setup information
- Location on the Desktop.

Each object class defines a set of keynames that can be used to control the attributes and behavior of its objects. An object's setup

information is processed by wpSetup inherited from WPObject. The wpSetup method is called when the object is created by WinCreateObject. Every Workplace Shell class inherits the set of keynames defined for the WPObject class, and can define additional ones that are unique to that class. Keynames and the values supported by the WPObject class are listed in the "Workplace Shell Setup Strings" subsection of the "Object Initialization and Termination: Setup/Cleanup Methods" section.

Newly created objects need to be placed somewhere on the Desktop, either directly on the Desktop or in a folder. All Workplace Shell objects have object IDs associated with them. The following table shows the object IDs associated with system folders. When an object is created by WinCreateObject, an application can specify its location as one of the predefined system folders.

```
ID              System Folder

WP_CONFIG       System setup folder.

WP_DESKTOP      Desktop.

WP_DRIVE        Drives folder.

WP_INFO         Information folder.

WP_NOWHERE      Hidden folder.

WP_START        Startup folder.

WP_SYSTEM       System folder.

WP_TEMPS        Templates folder.
```

WinCreateObject returns a handle to the newly created object. This handle is persistent and can be used at any time to reference the object. Applications that did not create the object can get a handle to the object by calling WinQueryObject. WinQueryObject requires either a full path name for a file object or an object ID.

Applications that get a handle to an object can change the behavior or state of that object by calling WinSetObjectData. This function results in the Workplace Shell calling the object's wpSetup method. By specifying a value for keyname variables defined for the object's class, an application, can effect changes to objects that already exist on the Desktop. The process of effecting changes to existing objects on the Desktop is summarized in the following figure:

```
/* Use ObjectID to get the handle to the object */
WinQueryObject(...);

/* Put keyname values in a setup string */
pszSetupString="KEYNAME1=value;...";

/* Change the behavior of an existing object */
WinSetObjectData(...);
```

Object Class Functions

For the Workplace Shell to know how to manipulate objects on the user's behalf, it must know about the object class and the class definition, that is, its data and methods. Therefore, object classes must be registered with the Workplace Shell. The following functions enable applications to affect object classes registered with the Workplace Shell:

- WinEnumObjectClasses
- WinRegisterObjectClass
- WinDeregisterObjectClass
- WinReplaceObjectClass.

WinEnumObjectClasses enables an application to get the list of all Workplace Shell object classes that have been registered with the Workplace Shell.

WinRegisterObjectClass and WinDeregisterObjectClass permit an application to register and deregister Workplace Shell object classes with the Workplace Shell. WinRegisterObjectClass registers the specified Workplace Shell object class that is defined in the specified DLL module that was created using the SOM compiler. The name of the object class must match the library name specified in the DEF file used to build the DLL. Because all registered classes are maintained in the OS2.INI file and are cached upon system initialization, a class should be

removed, if it is no longer needed.

WinReplaceObjectClass is used to replace a registered object class with another class. The replacement class is a subclass of the class being replaced. This type of class is useful for modifying the behavior of instances of a Workplace Shell object class without the instances being aware of the new class. WinReplaceObjectClass also is used to undo the replacement and restore the original class that defines the behavior of its instances.

Using Workplace Shell Win Functions

This section describes, through the used of sample code fragments, how to use Workplace Shell Win functions. Specifically it includes how to:

- Create objects
- Obtain the directory specification of a given object's handle
- Obtain the directory specification of the active Desktop
- Open objects
- Set objects data
- Save objects
- Save the current size and position of a window
- Move objects
- Copy objects
- Create a shadow of drive A on the Desktop
- Register and deregister objects
- Enumerate all classes registered with the system
- Replace a registered class with another
- Obtain the icon associated with a file and display it
- Destroy objects
- Shut down the system

Creating Objects

The following sample code shows how to add a new object of class WPFolder to the Desktop container:

```
#define INCL_WINWORKPLACE
#include "os2.h"

PSZ      pszObjTitle = "Sample Object";
HOBJECT  hobj;

{
  hobj = WinCreateObject("WPFolder",
                         pszObjTitle,
                         "OBJECTID=<WP_SAMPLE>",
                         <WP_DESKTOP>,
                         CO_FAILIFEXISTS);

  if (hobj != NULLHANDLE)
  {
    /* Object was successfully created on the Desktop */
  }
  else
```

```
  {
    /* WinCreateObject failed */
   }
}
```

Obtaining the Directory Specification for a Given Object's Handle

The following sample code shows how to obtain the directory specification of a given object's handle:

```
#define INCL_WINWORKPLACE
#include "os2.h"

HOBJECT hObject;
CHAR    szPath[CCHMAXPATH + 1];
BOOL    fSuccess;

hObject = WinQueryObject("<WP_OS2SYS>");
if (hObject != NULL)
{
  /* WinQueryObject was successful */
  fSuccess = WinQueryObjectPath(hObject,szPath,sizeof(szPath));

  if (fSuccess)
  {
    /* WinQueryObjectPath was successful */
  }
  else
  {
    /* WinQueryObjectPath failed */
  }
}
else
{
  /* WinQueryObject fails */
}
```

Obtaining the Directory Specification of the Active Desktop

The following sample code shows how to obtain the directory specification of the active Desktop:

```
#define INCL_WINWORKPLACE
#include "os2.h"

CHAR    szPath[CCHMAXPATH + 1];
BOOL    fSuccess;

fSuccess = WinQueryActiveDesktopPathname(szPath,sizeof(szPath));
if (fSuccess)
{
  /* WinQueryActiveDesktopPathname was successful */
}
else
{
  /* WinQueryActiveDesktopPathname failed */
}
```

Opening Objects

The following sample code shows how to open an object in the Settings notebook:

```
#define INCL_WINWORKPLACE
#include "os2.h"

HOBJECT hObject;
ULONG   ulView;
BOOL    fFlags
BOOL    fSuccess;

hObject = WinQueryObject("<WP_DESKTOP>");
if (hObject != NULL)
{
  /* WinQueryObject was successful */
  fSuccess = WinOpenObject(hObject,OPEN_SETTINGS,TRUE);

  if (fSuccess)
  {
    /* If concurrent views are off and a Settings notebook of the
     * Desktop already exists, then it was successfully resurfaced.
     *
     * If concurrent views are on, then another Settings notebook
     * of the Desktop was successfully opened.
     */
  }
  else
  {
    /* WinOpenObject failed */
  }
}
```

Setting Objects Data

The following sample code shows how to set the object data for the object with an ID of WP_SAMPLE, to have the specified title and use the specified help panel:

```
HOBJECT hObj;
BOOL    rc;

{
  hobj = WinQueryObject( "<WP_SAMPLE>" );
  if (hObj != NULLHANDLE)
  {
    /* WinQueryObject of WP_SAMPLE was successful */
    rc = WinSetObjectData(hObj,
                          "TITLE="Sample Object;
                          HELPPANEL=SAMPLEHELP_ID");
    if (rc)
    {
      /* Object data is set */
    }
    else
    {
      /* WinSetObjectData failed */
    }
  }
}
```

Saving Objects

The following sample code shows how to save the state of an object asynchronously:

```
#define INCL_WINWORKPLACE
#include "os2.h"

HOBJECT  Object;
BOOL     fAsync = TRUE;
BOOL     fSuccess

hObject = WinQueryObject("<WP_DESKTOP>");
if (hObject != NULL)
{
  /* WinQueryObject was successful */
  fSuccess = WinSaveObject(hObject,fAsync);

  if (fSuccess)
  {
    /* The state of the Desktop was saved asynchronously */
  }
  else
  {
    /* Asynchronous save of the Desktop failed */
  }
}
```

Saving the Current Size and Position of a Window

Knowing the application name, keyname and window handle, the following sample code fragment shows how to save the current size and position of the given window:

```
#define INCL_WINWORKPLACE
#include "os2.h"

{
  BOOL rc;

  rc = WinStoreWindowPos(vszAppName,szKey,hwnd);
  if (rc)
    /* Window size and position saved */
  else
    /* ERROR occurred */

/* Later you would use WinRestoreWindowPos to restore the window */
/* to it's previously saved size and position                     */

  .
  .
  .

rc = WinRestoreWindowPos(vszAppName,szKey,hwnd);
if (rc)
  /* Window size and position restored */
else
  /* ERROR occurred */
}
```

Moving Objects

The following sample code shows how to move the Drives object onto the Desktop:

```
#define INCL_WINWORKPLACE
#include "os2.h"

HOBJECT    hObjectofDest;
HOBJECT    hObjectofObject;
HOBJECT    hObjectofResult;

hObjectofObject = WinQueryObject("<WP_DRIVES>");
if (hObjectofObject != NULL)
{
  /* WinQueryObject of Drives was successful */
  hObjectofDest = WinQueryObject("<WP_DESKTOP>");

  if (hObjectofDest != NULL)
  {
    /* WinQueryObject of Startup was successful */
    hObjectofResult = WinMoveObject(hObjectofObject,
                                    hObjectofDest,
                                    0);

    if (hObjectofResult != NULL)
    {
      /* Drives object was successfully moved to the Desktop */
    }
    else
    {
      /* Move failed */
    }
  }
}
```

Copying Objects

The following sample code shows how to copy the drives object into the Startup folder, so the Drives folder will be opened during system startup:

```
#define INCL_WINWORKPLACE
#include "os2.h"

HOBJECT    hObjectofDest;
HOBJECT    hObjectofObject;
HOBJECT    hObjectofResult;

hObjectofObject = WinQueryObject("<WP_DRIVES>");
if (hObjectofObject != NULL)
{
  /* WinQueryObject of Drives was successful */
  hObjectofDest = WinQueryObject("<WP_START>");

  if (hObjectofDest != NULL)
  {
    /* WinQueryObject of Startup was successful */
    hObjectofResult = WinCopyObject(hObjectofObject,
                                    hObjectofDest,
                                    0);

    if (hObjectofResult != NULL)
    {
      /* Drives object was successfully moved to the Startup folder */
    }
    else
    {
      /* Copy failed */
    }
  }
}
```

Creating a Shadow of Drive A on the Desktop

The following sample code fragment shows how to create a shadow of drive A on the desktop:

```
#define INCL_WINWORKPLACE
#include "os2.h"

HOBJECT   hObjectofObject;
HOBJECT   hObjecttoDest;
HOBJECT   hObjectofResult;
{
  hObjectofObject = WinQueryObject("<WP_DOS_DRV_A>");
  if (hObjectofObject != NULL)
  {
    /* WinQueryObject of DOS from Dive A: object was successful */
    hObjecttoDest = WinQueryObject("<WP_DESKTOP>");
    if (hObjecttoDest != NULL)
    {
      /* WinQueryObject of DOS from Dive A: object was successful */
      hObjectofResult = WinCreateShadow(hObjectofObject,hObjecttoDest,NULL);
      if (hObjectofResult != NULL)
      {
        /* Shadow of DOS from Drive A: object is placed on the Desktop */
      }
      else
      {
        /* WinCreateShadow failed */
      }
    }
  }
}
```

Registering and Deregistering Objects

The following sample code shows how to register a new class, WPSample, with the Workplace Shell and later shows how to deregister it:

```
#define INCL_WINWORKPLACE
#include "os2.h"

PSZ  pszClassName="WPSample";
PSZ  pszModName="test.dll";
BOOL fReturn;

{
  fReturn = WinRegisterObjectClass(pszClassName, pszModName);
  if (fReturn)
  {
    /* WPSample class was successfully registered */
  }
  else
  {
    /* WinRegisterObjectClass failed */
   }
   .
   .
   .

   /* Later you would want to deregister the object */
   fReturn = WinDeregisterObjectClass(pszClassName);
   if (fReturn)
   {
     /* WPSample class was successfully deregistered */
   }
   else
   {
     /* WinDeregisterObjectClass failed */
```

```
    }
}
```

Enumerating All Classes Registered with the System

The following sample code shows how to enumerate all classes registered with the system:

```
#define INCL_WINWORKPLACE
#include "os2.h"

PULONG    ulSize;
POBJCLASS pObjClass;
BOOL      rc;

{
  /* Get the buffer size required to hold all registered classes */
  if (WinEnumObjectClasses(NULL, &ulSize)
     &&
     (pmem = (PVOID) AllocMem (ulSize, NULL)))
  {
    pObjClass = (POBJCLASS)pmem;
    rc = WinEnumObjectClasses(pObjClass, &ulSize)
    if (rc)
    {
      /* pObjClass contains all classes registered with the system */
    }
    else
    {
      /* WinEnumObjectClasses failed */
    }
}
```

Replacing a Registered Class with Another Registered Class

The following sample code shows how to replace a WPFolder object class with a WPMyFolder object class at restart of the system:

```
#define INCL_WINWORKPLACE
#include "os2.h"

PSZ  pszNewClassName="WPMyFolderClass";
PSZ  pszOldClassName="WPFolder";
BOOL rc;

{
  rc = WinReplaceObjectClass(pszOldClassName, pszNewClassName, TRUE);
  if (rc)
  {
    /* Object will be replaced at next start of the system */
  }
  else
  {
    /* WinReplaceObjectClass failed */
   }
}
```

Obtaining the Icon Associated with a File and Displaying it

The following sample code fragment shows how to obtain the icon associated with the TUTORIAL.EXE file and shows how to display it as the user's tutorial application:

```
#define  INCL_WINWORKPLACE
#include "os2.h"

PSZ pszFileName = "tutorial.exe";
PSZ pszMyTutorial = "mytutor.exe";

HPOINTER hptr;
ULONG    rc;

{
  hptr = WinLoadFileIcon(pszFileName, FALSE);
  if (hptr != NULL)
  {
    /* Obtain the icon associated with TUTORIAL.EXE for displaying purpose only */
    rc = WinSetFileIcon(pszMyTutorial, hptr);
    if (rc != NULL)
    {
      /* MYTUTOR.EXE now has the icon associated with TUTORIAL.EXE */
    }
    else
    {
      /* WinSetFileIcon failed */
    }
  }
  else
  {
     /* WinLoadFileIcon failed */
  }

   .
   .
   .
  /* Later free the file icon associated with the TUTORIAL.EXE file */
  rc = WinFreeFileIcon(hptr);
  if (rc != NULL)
  {
    /* Icon pointer freed */
  }
  else
  {
    /* WinFreeFileIcon failed */
  }
}
```

Destroying Objects

The following sample code shows how to destroy the object associated with the OBJECTID of the WP_SAMPLE class, registered earlier:

```
#define INCL_WINWORKPLACE
#include "os2.h"

HOBJECT hObj;
BOOL    rc;

{
  hobj = WinQueryObject( "<WP_SAMPLE>" );
  if (hObj != NULLHANDLE)
  {
    /* WinQueryObject of WP_SAMPLE was successful */
    rc = WinDestroyObject(hObj);
    if (rc)
      /* hObj was successfully destroyed */
    else
      /* WinDestroyObject failed */
  }
  else
```

```
  {
    /* WinQueryObject failed */
    return;
  }
}
```

Shutting Down the System

Given the window handle and the message queue handle, the following code fragment shows how to shut down the system:

```
BOOL rc;
HAB  hab;
HMQ  hmq;

{
  hab = WinQueryAnchorBlock(hwnd);
  rc = WinShutDownSystem(hab, hmq);
  if (rc)
  {
    /* System shutdown successfully */
  }
  else
  {
    /* WinShutDownSystem failed */
   }
}
```

Notices

Second Edition (January 1996)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as

follows: "(C) (your company name) (year). All rights reserved."

(C) Copyright International Business Machines Corporation 1994, 1996. All rights reserved.
Note to U.S. Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Common User Access	CUA
Operating System/2	OS/2
PM	Presentation Manager
SAA	Systems Application Architecture
SOMobjects	IBM
Workplace Shell	

The following terms are trademarks of other companies:

C++	American Telephone and Telegraph Company
CORBA	Object Management Group, Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

This glossary defines many of the terms used in this book. It includes terms and definitions from the *IBM Dictionary of Computing* , as well as terms specific to the OS/2 operating system and the Presentation Manager. It is not a complete glossary for the entire OS/2 operating system; nor is it a complete dictionary of computer terms.

Other primary sources for these definitions are:

- The *American National Standard Dictionary for Information Systems* , ANSI X3.172-1990, copyrighted 1990 by the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. These definitions are identified by the symbol (A) after the definition.

- The *Information Technology Vocabulary* , developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

Glossary Listing

Select a starting letter of glossary terms:

A N
B O
C P
D Q
E R
F S
G T
H U
I V
J W
K X
L Y
M Z

Glossary - A

accelerator -In SAA Common User Access architecture, a key or combination of keys that invokes an application-defined function.

accelerator table -A table used to define which key strokes are treated as *accelerators* and the commands they are translated into.

access mode -The manner in which an application gains access to a file it has opened. Examples of access modes are read-only, write-only, and read/write.

access permission -All access rights that a user has regarding an object. (I)

action -One of a set of defined tasks that a computer performs. Users request the application to perform an action in several ways, such as typing a command, pressing a function key, or selecting the action name from an action bar or menu.

action bar -In SAA Common User Access architecture, the area at the top of a window that contains choices that give a user access to actions available in that window.

action point -The current position on the screen at which the pointer is pointing. Contrast with *hot spot* and *input focus* .

active program -A program currently running on the computer. An active program can be interactive (running and receiving input from the user) or noninteractive (running but not receiving input from the user). See also *interactive program* and *noninteractive program* .

active window -The window with which the user is currently interacting.

address space -(1) The range of addresses available to a program. (A) (2) The area of virtual storage available for a particular job.

alphanumeric video output -Output to the logical video buffer when the video adapter is in text mode and the logical video buffer is addressed by an application as a rectangular array of character cells.

American National Standard Code for Information Interchange -The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

Note: IBM has defined an extension to ASCII code (characters 128-255).

anchor -A window procedure that handles Presentation Manager message conversions between an icon procedure and an application.

anchor block -An area of Presentation-Manager-internal resources to allocated process or thread that calls WinInitialize.

anchor point -A point in a window used by a program designer or by a window manager to position a subsequently appearing window.

ANSI -American National Standards Institute.

APA -All points addressable.

API -Application programming interface.

application -A collection of software components used to perform specific types of work on a computer; for example, a payroll application, an airline reservation application, a network application.

application object -In SAA Advanced Common User Access architecture, a form that an application provides for a user; for example, a spreadsheet form. Contrast with *user object* .

application programming interface (API) -A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

application-modal -Pertaining to a message box or dialog box for which processing must be completed before further interaction with any other window owned by the same application may take place.

area -In computer graphics, a filled shape such as a solid rectangle.

ASCII -American National Standard Code for Information Interchange.

ASCIIZ -A string of ASCII characters that is terminated with a byte containing the value 0.

aspect ratio -In computer graphics, the width-to-height ratio of an area, symbol, or shape.

asynchronous (ASYNC) -(1) Pertaining to two or more processes that do not depend upon the occurrence of specific events such as common timing signals. (T) (2) Without regular time relationship; unexpected or unpredictable with respect to the execution of program instructions. See also *synchronous* .

atom -A constant that represents a string. As soon as a string has been defined as an atom, the atom can be used in place of the string to save space. Strings are associated with their respective atoms in an *atom table* . See also *integer atom* .

atom table -A table used to relate *atoms* with the strings that they represent. Also in the table is the mechanism by which the presence of a string can be checked.

atomic operation -An operation that completes its work on an object before another operation can be performed on the same object.

attribute -A characteristic or property that can be controlled, usually to obtain a required appearance; for example, the color of a line. See also *graphics attributes* and *segment attributes* .

automatic link -In Information Presentation Facility (IPF), a link that begins a chain reaction at the primary window. When the user selects the primary window, an automatic link is activated to display secondary windows.

AVIO -Advanced Video Input/Output.

Glossary - B

Bézier curve -(1) A mathematical technique of specifying smooth continuous lines and surfaces, which require a starting point and a finishing point with several intermediate points that influence or control the path of the linking curve. Named after Dr. P. Bézier. (2) (D of C) In the AIX Graphics Library, a cubic spline approximation to a set of four control points that passes through the first and fourth control points and that has a continuous slope where two spline segments meet. Named after Dr. P. Bézier.

background -(1) In multiprogramming, the conditions under which low-priority programs are executed. Contrast with *foreground* . (2) An active session that is not currently displayed on the screen.

background color -The color in which the background of a graphic primitive is drawn.

background mix -An attribute that determines how the background of a graphic primitive is combined with the existing color of the graphics presentation space. Contrast with *mix* .

background program -In multiprogramming, a program that executes with a low priority. Contrast with *foreground program* .

bit map -A representation in memory of the data displayed on an APA device, usually the screen.

block -(1) A string of data elements recorded or transmitted as a unit. The elements may be characters, words, or logical records. (T) (2) To

record data in a block. (3) A collection of contiguous records recorded as a unit. Blocks are separated by interblock gaps and each block may contain one or more records. (A)

block device -A storage device that performs I/O operations on blocks of data called *sectors* . Data on block devices can be randomly accessed. Block devices are designated by a drive letter (for example, **C:**).

blocking mode -A condition set by an application that determines when its threads might block. For example, an application might set the Pipemode parameter for the DosCreateNPipe function so that its threads perform I/O operations to the named pipe block when no data is available.

border -A visual indication (for example, a separator line or a background color) of the boundaries of a window.

boundary determination -An operation used to compute the size of the smallest rectangle that encloses a graphics object on the screen.

breakpoint -(1) A point in a computer program where execution may be halted. A breakpoint is usually at the beginning of an instruction where halts, caused by external intervention, are convenient for resuming execution. (T) (2) A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

broken pipe -When all of the handles that access one end of a pipe have been closed.

bucket -One or more fields in which the result of an operation is kept.

buffer -(1) A portion of storage used to hold input or output data temporarily. (2) To allocate and schedule the use of buffers. (A)

button -A mechanism used to request or initiate an action. See also *barrel buttons* , *bezel buttons* , *mouse button* , *push button* , and *radio button* .

byte pipe -Pipes that handle data as byte streams. All unnamed pipes are byte pipes. Named pipes can be byte pipes or message pipes. See *byte stream* .

byte stream -Data that consists of an unbroken stream of bytes.

Glossary - C

cache -A high-speed buffer storage that contains frequently accessed instructions and data; it is used to reduce access time.

cached micro presentation space -A presentation space from a Presentation-Manager-owned store of micro presentation spaces. It can be used for drawing to a window only, and must be returned to the store when the task is complete.

CAD -Computer-Aided Design.

call -(1) The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (I) (A) (2) To transfer control to a procedure, program, routine, or subroutine.

calling sequence -A sequence of instructions together with any associated data necessary to execute a call. (T)

Cancel -An action that removes the current window or menu without processing it, and returns the previous window.

cascaded menu -In the OS/2 operating system, a menu that appears when the arrow to the right of a cascading choice is selected. It contains a set of choices that are related to the cascading choice. Cascaded menus are used to reduce the length of a menu. See also *cascading choice* .

cascading choice -In SAA Common User Access architecture, a choice in a menu that, when selected, produces a cascaded menu containing other choices. An arrow () appears to the right of the cascading choice.

CASE statement -In PM programming, provides the body of a window procedure. There is usually one CASE statement for each message type supported by an application.

CGA -Color graphics adapter.

chained list -A list in which the data elements may be dispersed but in which each data element contains information for locating the next. (T) Synonymous with *linked list* .

character -A letter, digit, or other symbol.

character box -In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single character from a character set. See also *character mode* . Contrast with *character cell* .

character cell -The physical, rectangular space in which any single character is displayed on a screen or printer device. Position is addressed

by row and column coordinates. Contrast with *character box* .

character code -The means of addressing a character in a character set, sometimes called *code point* .

character device -A device that performs I/O operations on one character at a time. Because character devices view data as a stream of bytes, character-device data cannot be randomly accessed. Character devices include the keyboard, mouse, and printer, and are referred to by name.

character mode -A mode that, in conjunction with the font type, determines the extent to which graphics characters are affected by the character box, shear, and angle attributes.

character set -(1) An ordered set of unique representations called characters; for example, the 26 letters of English alphabet, Boolean 0 and 1, the set of symbols in the Morse code, and the 128 ASCII characters. (A) (2) All the valid characters for a programming language or for a computer system. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print.

check box -In SAA Advanced Common User Access architecture, a square box with associated text that represents a choice. When a user selects a choice, an X appears in the check box to indicate that the choice is in effect. The user can clear the check box by selecting the choice again. Contrast with *radio button* .

check mark - (1) (D of C) In SAA Advanced Common User Access architecture, a symbol that shows that a choice is currently in effect. (2) The symbol that is used to indicate a selected item on a pull-down menu.

child process -In the OS/2 operating system, a process started by another process, which is called the parent process. Contrast with *parent process* .

child window -A window that appears within the border of its parent window (either a primary window or another child window). When the parent window is resized, moved, or destroyed, the child window also is resized, moved, or destroyed; however, the child window can be moved or resized independently from the parent window, within the boundaries of the parent window. Contrast with *parent window* .

choice -(1) An option that can be selected. The choice can be presented as text, as a symbol (number or letter), or as an icon (a pictorial symbol). (2) (D of C) In SAA Common User Access architecture, an item that a user can select.

chord -(1) To press more than one button on a pointing device while the pointer is within the limits that the user has specified for the operating environment. (2) (D of C) In graphics, a short line segment whose end points lie on a circle. Chords are a means for producing a circular image from straight lines. The higher the number of chords per circle, the smoother the circular image.

class -A way of categorizing objects based on their behavior and shape. A class is, in effect, a definition of a generic object. In SOM, a class is a special kind of object that can manufacture other objects that all have a common shape and exhibit similar behavior (more precisely, all of the objects manufactured by a class have the same memory layout and share a common set of methods). New classes can be defined in terms of existing classes through a technique known as *inheritance* .

class method -A class method of class <X> is a method provided by the metaclass of class <X>. Class methods are executed without requiring any instances of class <X> to exist, and are frequently used to create instances. In System Object Model, an action that can be performed on a class object.

class object -In System Object Model, the run-time implementation of a class.

class style -The set of properties that apply to every window in a window class.

client -(1) A functional unit that receives shared services from a server. (T) (2) A user, as in a client process that uses a named pipe or queue that is created and owned by a server process.

client area -The part of the window, inside the border, that is below the menu bar. It is the user's work space, where a user types information and selects choices from selection fields. In primary windows, it is where an application programmer presents the objects that a user works on.

client program -An application that creates and manipulates instances of classes.

client window -The window in which the application displays output and receives input. This window is located inside the frame window, under the window title bar and any menu bar, and within any scroll bars.

clip limits -The area of the paper that can be reached by a printer or plotter.

clipboard -In SAA Common User Access architecture, an area of computer memory, or storage, that temporarily holds data. Data in the clipboard is available to other applications.

clipping -In computer graphics, removing those parts of a display image that lie outside a given boundary. (I) (A)

clipping area -The area in which the window can paint.

clipping path -A clipping boundary in world-coordinate space.

clock tick -The minimum unit of time that the system tracks. If the system timer currently counts at a rate of X Hz, the system tracks the time every 1/X of a second. Also known as *time tick* .

CLOCK$ -Character-device name reserved for the system clock.

code page -An assignment of graphic characters and control-function meanings to all code points.

code point -(1) Synonym for *character code* . (2) (D of C) A 1-byte code representing one of 256 potential characters.

code segment -An executable section of programming code within a load module.

color dithering -See *dithering* .

color graphics adapter (CGA) -An adapter that simultaneously provides four colors and is supported by all IBM Personal Computer and Personal System/2 models.

command -The name and parameters associated with an action that a program can perform.

command area -An area composed of a command field prompt and a command entry field.

command entry field -An entry field in which users type commands.

command line -On a display screen, a display line, sometimes at the bottom of the screen, in which only commands can be entered.

command mode -A state of a system or device in which the user can enter commands.

command prompt -A field prompt showing the location of the command entry field in a panel.

Common Programming Interface (CPI) -Definitions of those application development languages and services that have, or are intended to have, implementations on and a high degree of commonality across the SAA environments. One of the three SAA architectural areas. See also *Common User Access architecture* .

Common User Access (CUA) architecture - Guidelines for the dialog between a human and a workstation or terminal. One of the three SAA architectural areas. See also *Common Programming Interface* .

compile -To translate a program written in a higher-level programming language into a machine language program.

composite window -A window composed of other windows (such as a frame window, frame-control windows, and a client window) that are kept together as a unit and that interact with each other.

computer-aided design (CAD) -The use of a computer to design or change a product, tool, or machine, such as using a computer for drafting or illustrating.

COM1, COM2, COM3 -Character-device names reserved for serial ports 1 through 3.

CON -Character-device name reserved for the console keyboard and screen.

conditional cascaded menu -A pull-down menu associated with a menu item that has a cascade mini-push button beside it in an object's pop-up menu. The conditional cascaded menu is displayed when the user selects the mini-push button.

container -In SAA Common User Access architecture, an object that holds other objects. A folder is an example of a container object. See also *folder* and *object* .

contextual help -In SAA Common User Access Architecture, help that gives specific information about the item the cursor is on. The help is contextual because it provides information about a specific item as it is currently being used. Contrast with *extended help* .

contiguous -Touching or joining at a common edge or boundary, for example, an unbroken consecutive series of storage locations.

control -In SAA Advanced Common User Access architecture, a component of the user interface that allows a user to select choices or type information; for example, a check box, an entry field, a radio button.

control area -A storage area used by a computer program to hold control information. (I) (A)

Control Panel -In the Presentation Manager, a program used to set up user preferences that act globally across the system.

Control Program -(1) The basic functions of the operating system, including DOS emulation and the support for keyboard, mouse, and video input/output. (2) A computer program designed to schedule and to supervise the execution of programs of a computer system. (I) (A)

control window -A window that is used as part of a composite window to perform simple input and output tasks. Radio buttons and check boxes are examples.

control word -An instruction within a document that identifies its parts or indicates how to format the document.

coordinate space -A two-dimensional set of points used to generate output on a video display of printer.

Copy -A choice that places onto the clipboard, a copy of what the user has selected. See also *Cut* and *Paste* .

correlation -The action of determining which element or object within a picture is at a given position on the display. This follows a *pick* operation.

coverpage window -A window in which the application's help information is displayed.

CPI -Common Programming Interface.

critical extended attribute -An extended attribute that is necessary for the correct operation of the system or a particular application.

critical section -(1) In programming languages, a part of an asynchronous procedure that cannot be executed simultaneously with a certain part of another asynchronous procedure. (I)

Note: Part of the other asynchronous procedure also is a critical section. (2) A section of code that is not reentrant; that is, code that can be executed by only one thread at a time.

CUA architecture -Common User Access architecture.

current position -In computer graphics, the position, in user coordinates, that becomes the starting point for the next graphics routine, if that routine does not explicitly specify a starting point.

cursor -A symbol displayed on the screen and associated with an input device. The cursor indicates where input from the device will be placed. Types of cursors include text cursors, graphics cursors, and selection cursors. Contrast with *pointer* and *input focus* .

Cut -In SAA Common User Access architecture, a choice that removes a selected object, or a part of an object, to the clipboard, usually compressing the space it occupied in a window. See also *Copy* and *Paste* .

Glossary - D

daisy chain -A method of device interconnection for determining interrupt priority by connecting the interrupt sources serially.

data segment -A nonexecutable section of a program module; that is, a section of a program that contains data definitions.

data structure -The syntactic structure of symbolic expressions and their storage-allocation characteristics. (T)

data transfer -The movement of data from one object to another by way of the clipboard or by direct manipulation.

DBCS -Double-byte character set.

DDE -Dynamic data exchange.

deadlock -(1) Unresolved contention for the use of a resource. (2) An error condition in which processing cannot continue because each of two elements of the process is waiting for an action by, or a response from, the other. (3) An impasse that occurs when multiple processes are waiting for the availability of a resource that will not become available because it is being held by another process that is in a similar wait state.

debug -To detect, diagnose, and eliminate errors in programs. (T)

decipoint -In printing, one tenth of a point. There are 72 points in an inch.

default procedure -A function provided by the Presentation Manager Interface that may be used to process standard messages from dialogs or windows.

default value -A value assumed when no value has been specified. Synonymous with assumed value. For example, in the graphics programming interface, the default line-type is 'solid'.

definition list -A type of list that pairs a term and its description.

delta -An application-defined threshold, or number of container items, from either end of the list.

descendant -See *child process* .

descriptive text -Text used in addition to a field prompt to give more information about a field.

Deselect all -A choice that cancels the selection of all of the objects that have been selected in that window.

Desktop Manager -In the Presentation Manager, a window that displays a list of groups of programs, each of which can be started or stopped.

desktop window -The window, corresponding to the physical device, against which all other types of windows are established.

detached process -A background process that runs independent of the parent process.

detent -A point on a slider that represents an exact value to which a user can move the slider arm.

device context -A logical description of a data destination such as memory, metafile, display, printer, or plotter. See also *direct device* context, *information device context* , *memory device context* , *metafile device context* , *queued device context* , and *screen device context* .

device driver -A file that contains the code needed to attach and use a device such as a display, printer, or plotter.

device space -(1) Coordinate space in which graphics are assembled after all GPI transformations have been applied. Device space is defined in device-specific units. (2) (D of C) In computer graphics, a space defined by the complete set of addressable points of a display device. (A)

dialog -The interchange of information between a computer and its user through a sequence of requests by the user and the presentation of responses by the computer.

dialog box -In SAA Advanced Common User Access architecture, a movable window, fixed in size, containing controls that a user uses to provide information required by an application so that it can continue to process a user request. See also *message box, primary window, secondary window* . Also known as a *pop-up window* .

Dialog Box Editor -A *WYSIWYG* editor that creates dialog boxes for communicating with the application user.

dialog item -A component (for example, a menu or a button) of a dialog box. Dialog items are also used when creating dialog templates.

dialog procedure -A dialog window that is controlled by a window procedure. It is responsible for responding to all messages sent to the dialog window.

dialog tag language -A markup language used by the DTL compiler to create dialog objects.

dialog template -The definition of a dialog box, which contains details of its position, appearance, and window ID, and the window ID of each of its child windows.

direct device context -A logical description of a data destination that is a device other than the screen (for example, a printer or plotter), and where the output is not to go through the spooler. Its purpose is to satisfy queries. See also *device context* .

direct manipulation -The user's ability to interact with an object by using the mouse, typically by dragging an object around on the Desktop and dropping it on other objects.

direct memory access (DMA) -A technique for moving data directly between main storage and peripheral equipment without requiring processing of the data by the processing unit.(T)

directory -A type of file containing the names and controlling information for other files or other directories.

display point -Synonym for *pel* .

dithering -(1) The process used in color displays whereby every other pel is set to one color, and the intermediate pels are set to another. Together they produce the effect of a third color at normal viewing distances. This process can only be used on solid areas of color; it does not work, for example, on narrow lines. (2) (D of C) In computer graphics, a technique of interleaving dark and light pixels so that the resulting image looks smoothly shaded when viewed from a distance.

DMA -Direct memory access.

DOS Protect Mode Interface (DPMI) -An interface between protect mode and real mode programs.

double-byte character set (DBCS) -A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more characters than can be represented by 256 code points, require double-byte character sets. Since each character requires two bytes, the entering, displaying, and printing of DBCS characters requires hardware and software that can support DBCS.

doubleword -A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. (A)

DPMI -DOS Protect Mode Interface.

drag -In SAA Common User Access, to use a pointing device to move an object; for example, clicking on a window border, and dragging it to make the window larger.

dragging -(1) In computer graphics, moving an object on the display screen as if it were attached to the pointer. (2) (D of C) In computer graphics, moving one or more segments on a display surface by translating. (I) (A)

drawing chain -See *segment chain* .

drop -To fix the position of an object that is being dragged, by releasing the select button of the pointing device. See also *drag* .

DTL -Dialog tag language.

dual-boot function -A feature of the OS/2 operating system that allows the user to start DOS from within the operating system, or an OS/2

session from within DOS.

duplex -Pertaining to communication in which data can be sent and received at the same time. Synonymous with *full duplex* .

dynamic data exchange (DDE) -A message protocol used to communicate between applications that share data. The protocol uses shared memory as the means of exchanging data between applications.

dynamic data formatting -A formatting procedure that enables you to incorporate text, bit maps or metafiles in an IPF window at execution time.

dynamic link library -A collection of executable programming code and data that is bound to an application at load time or run time, rather than during linking. The programming code and data in a dynamic link library can be shared by several applications simultaneously.

dynamic linking -The process of resolving external references in a program module at load time or run time rather than during linking.

dynamic segments -Graphics segments drawn in exclusive-OR mix mode so that they can be moved from one screen position to another without affecting the rest of the displayed picture.

dynamic storage -(1) A device that stores data in a manner that permits the data to move or vary with time such that the specified data is not always available for recovery. (A) (2) A storage in which the cells require repetitive application of control signals in order to retain stored data. Such repetitive application of the control signals is called a refresh operation. A dynamic storage may use static addressing or sensing circuits. (A) (3) See also *static storage* .

dynamic time slicing -Varies the size of the time slice depending on system load and paging activity.

dynamic-link module -A module that is linked at load time or run time.

Glossary - E

EBCDIC -Extended binary-coded decimal interchange code. A coded character set consisting of 8-bit coded characters (9 bits including parity check), used for information interchange among data processing systems, data communications systems, and associated equipment.

edge-triggered -Pertaining to an event semaphore that is posted then reset before a waiting thread gets a chance to run. The semaphore is considered to be posted for the rest of that thread's waiting period; the thread does not have to wait for the semaphore to be posted again.

EGA -Extended graphics adapter.

element -An entry in a graphics segment that comprises one or more graphics orders and that is addressed by the element pointer.

EMS -Expanded Memory Specification.

encapsulation -Hiding an object's implementation, that is, its private, internal data and methods. Private variables and methods are accessible only to the object that contains them.

entry field -In SAA Common User Access architecture, an area where a user types information. Its boundaries are usually indicated. See also *selection field* .

entry panel -A defined panel type containing one or more entry fields and protected information such as headings, prompts, and explanatory text.

entry-field control -The component of a user interface that provides the means by which the application receives data entered by the user in an entry field. When it has the input focus, the entry field displays a flashing pointer at the position where the next typed character will go.

environment segment -The list of environment variables and their values for a process.

environment strings -ASCII text strings that define the value of environment variables.

environment variables -Variables that describe the execution environment of a process. These variables are named by the operating system or by the application. Environment variables named by the operating system are PATH, DPATH, INCLUDE, INIT, LIB, PROMPT, and TEMP. The values of environment variables are defined by the user in the CONFIG.SYS file, or by using the SET command at the OS/2 command prompt.

error message -An indication that an error has been detected. (A)

event semaphore -A semaphore that enables a thread to signal a waiting thread or threads that an event has occurred or that a task has been completed. The waiting threads can then perform an action that is dependent on the completion of the signaled event.

exception -An abnormal condition such as an I/O error encountered in processing a data set or a file.

exclusive system semaphore -A system semaphore that can be modified only by threads within the same process.

executable file -(1) A file that contains programs or commands that perform operations or actions to be taken. (2) A collection of related data records that execute programs.

exit -To execute an instruction within a portion of a computer program in order to terminate the execution of that portion. Such portions of computer programs include loops, subroutines, modules, and so on. (T) Repeated exit requests return the user to the point from which all functions provided to the system are accessible. Contrast with *cancel* .

expanded memory specification (EMS) -Enables DOS applications to access memory above the 1MB real mode addressing limit.

extended attribute -An additional piece of information about a file object, such as its data format or category. It consists of a name and a value. A file object may have more than one extended attribute associated with it.

extended help -In SAA Common User Access architecture, a help action that provides information about the contents of the application window from which a user requested help. Contrast with *contextual help* .

extended-choice selection -A mode that allows the user to select more than one item from a window. Not all windows allow extended choice selection. Contrast with *multiple-choice selection* .

extent -Continuous space on a disk or diskette that is occupied by or reserved for a particular data set, data space, or file.

external link -In Information Presentation Facility, a link that connects external online document files.

Glossary - F

family-mode application -An application program that can run in the OS/2 environment and in the DOS environment; however, it cannot take advantage of many of the OS/2-mode facilities, such as multitasking, interprocess communication, and dynamic linking.

FAT -File allocation table.

FEA -Full extended attribute.

field-level help -Information specific to the field on which the cursor is positioned. This help function is "contextual" because it provides information about a specific item as it is currently used; the information is dependent upon the context within the work session.

FIFO -First-in-first-out. (A)

file -A named set of records stored or processed as a unit. (T)

file allocation table (FAT) -In IBM personal computers, a table used by the operating system to allocate space on a disk for a file, and to locate and chain together parts of the file that may be scattered on different sectors so that the file can be used in a random or sequential manner.

file attribute -Any of the attributes that describe the characteristics of a file.

File Manager -In the Presentation Manager, a program that displays directories and files, and allows various actions on them.

file specification -The full identifier for a file, which includes its drive designation, path, file name, and extension.

file system -The combination of software and hardware that supports storing information on a storage device.

file system driver (FSD) -A program that manages file I\O and controls the format of information on the storage media.

fillet -A curve that is tangential to the end points of two adjoining lines. See also *polyfillet* .

filtering -An application process that changes the order of data in a queue.

first-in-first-out (FIFO) -A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time. (A)

flag -(1) An indicator or parameter that shows the setting of a switch. (2) A character that signals the occurrence of some condition, such as the end of a word. (A) (3) (D of C) A characteristic of a file or directory that enables it to be used in certain ways. See also *archive flag* , *hidden flag* , and *read-only flag* .

focus -See *input focus* .

folder -A container used to organize objects.

font -A particular size and style of typeface that contains definitions of character sets, marker sets, and pattern sets.

Font Editor -A utility program provided with the IBM Developers Toolkit that enables the design and creation of new fonts.

foreground program -(1) The program with which the user is currently interacting. Also known as *interactive program* . Contrast with *background program* . (2) (D of C) In multiprogramming, a high-priority program.

frame -The part of a window that can contain several different visual elements specified by the application, but drawn and controlled by the Presentation Manager. The frame encloses the client area.

frame styles -Standard window layouts provided by the Presentation Manager.

FSD -File system driver.

full-duplex -Synonym for *duplex* .

full-screen application -An application that has complete control of the screen.

function -(1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a call. (2) A set of related control statements that cause one or more programs to be performed.

function key -A key that causes a specified sequence of operations to be performed when it is pressed, for example, F1 and Alt-K.

function key area -The area at the bottom of a window that contains function key assignments such as F1=Help.

Glossary - G

GDT -Global Descriptor Table.

general protection fault -An exception condition that occurs when a process attempts to use storage or a module that has some level of protection assigned to it, such as I/O privilege level. See also *IOPL code segment* .

Global Descriptor Table (GDT) -A table that defines code and data segments available to all tasks in an application.

global dynamic-link module -A dynamic-link module that can be shared by all processes in the system that refer to the module name.

global file-name character -Either a question mark (?) or an asterisk (*) used as a variable in a file name or file name extension when referring to a particular file or group of files.

glyph -A graphic symbol whose appearance conveys information.

GPI -Graphics programming interface.

graphic primitive -In computer graphics, a basic element, such as an arc or a line, that is not made up of smaller parts and that is used to create diagrams and pictures. See also *graphics segment* .

graphics -(1) A picture defined in terms of graphic primitives and graphics attributes. (2) (D of C) The making of charts and pictures. (3) Pertaining to charts, tables, and their creation. (4) See *computer graphics, coordinate graphics, fixed-image graphics, interactive graphics, passive graphics, raster graphics* .

graphics attributes -Attributes that apply to graphic primitives. Examples are color, line type, and shading-pattern definition. See also *segment attributes* .

graphics field -The clipping boundary that defines the visible part of the presentation-page contents.

graphics mode -One of several states of a display. The mode determines the resolution and color content of the screen.

graphics model space -The conceptual coordinate space in which a picture is constructed after any model transforms have been applied. Also known as *model space* .

Graphics programming interface -The formally defined programming language that is between an IBM graphics program and the user of the program.

graphics segment -A sequence of related graphic primitives and graphics attributes. See also *graphic primitive* .

graying -The indication that a choice on a pull-down is unavailable.

group -A collection of logically connected controls. For example, the buttons controlling paper size for a printer could be called a group. See

also *program group* .

Glossary - H

handle -(1) An identifier that represents an object, such as a device or window, to the Presentation Interface. (2) (D of C) In the Advanced DOS and OS/2 operating systems, a binary value created by the system that identifies a drive, directory, and file so that the file can be found and opened.

hard error -An error condition on a network that requires either that the system be reconfigured or that the source of the error be removed before the system can resume reliable operation.

header -(1) System-defined control information that precedes user data. (2) The portion of a message that contains control information for the message, such as one or more destination fields, name of the originating station, input sequence number, character string indicating the type of message, and priority level for the message.

heading tags -A document element that enables information to be displayed in windows, and that controls entries in the contents window controls placement of push buttons in a window, and defines the shape and size of windows.

heap -An area of free storage available for dynamic allocation by an application. Its size varies according to the storage requirements of the application.

help function -(1) A function that provides information about a specific field, an application panel, or information about the help facility. (2) (D of C) One or more display images that describe how to use application software or how to do a system operation.

Help index -In SAA Common User Access architecture, a help action that provides an index of the help information available for an application.

help panel -A panel with information to assist users that is displayed in response to a help request from the user.

help window -A Common-User-Access-defined secondary window that displays information when the user requests help.

hidden file -An operating system file that is not displayed by a directory listing.

hide button -In the OS/2 operating system, a small, square button located in the right-hand corner of the title bar of a window that, when selected, removes from the screen all the windows associated with that window. Contrast with *maximize button* . See also *restore button* .

hierarchical inheritance -The relationship between parent and child classes. An object that is lower in the inheritance hierarchy than another object, inherits all the characteristics and behaviors of the objects above it in the hierarchy.

hierarchy -A tree of segments beginning with the root segment and proceeding downward to dependent segment types.

high-performance file system (HPFS) -In the OS/2 operating system, an installable file system that uses high-speed buffer storage, known as a cache, to provide fast access to large disk volumes. The file system also supports the coexistence of multiple, active file systems on a single personal computer, with the capability of multiple and different storage devices. File names used with the HPFS can have as many as 254 characters.

hit testing -The means of identifying which window is associated with which input device event.

hook -A point in a system-defined function where an application can supply additional code that the system processes as though it were part of the function.

hook chain -A sequence of hook procedures that are "chained" together so that each event is passed, in turn, to each procedure in the chain.

hot spot -The part of the pointer that must touch an object before it can be selected. This is usually the tip of the pointer. Contrast with *action point* .

HPFS -high-performance file system.

hypergraphic link -A connection between one piece of information and another through the use of graphics.

hypertext -A way of presenting information online with connections between one piece of information and another, called *hypertext links* . See also *hypertext link* .

hypertext link -A connection between one piece of information and another.

Glossary - I

I/O operation -An input operation to, or output operation from a device attached to a computer.

I-beam pointer -A pointer that indicates an area, such as an entry field in which text can be edited.

icon -In SAA Advanced Common User Access architecture, a graphical representation of an object, consisting of an image, image background, and a label. Icons can represent items (such as a document file) that the user wants to work on, and actions that the user wants to perform. In the Presentation Manager, icons are used for data objects, system actions, and minimized programs.

icon area -In the Presentation Manager, the area at the bottom of the screen that is normally used to display the icons for minimized programs.

Icon Editor -The Presentation Manager-provided tool for creating icons.

IDL -Interface Definition Language.

image font -A set of symbols, each of which is described in a rectangular array of pels. Some of the pels in the array are set to produce the image of one of the symbols. Contrast with *outline font* .

implied metaclass -Subclassing the metaclass of a parent class without a separate CSC for the resultant metaclass.

indirect manipulation -Interaction with an object through choices and controls.

information device context -A logical description of a data destination other than the screen (for example, a printer or plotter), but where no output will occur. Its purpose is to satisfy queries. See also *device context* .

information panel -A defined panel type characterized by a body containing only protected information.

Information Presentation Facility (IPF) -A facility provided by the OS/2 operating system, by which application developers can produce online documentation and context-sensitive online help panels for their applications.

inheritance -The technique of specifying the shape and behavior of one class (called a *subclass*) as incremental differences from another class (called the *parent class* or *superclass*). The subclass inherits the superclass' state representation and methods, and can provide additional data elements and methods. The subclass also can provide new functions with the same method names used by the superclass. Such a subclass method is said to override the superclass method, and will be selected automatically by method resolution on subclass instances. An overriding method can elect to call upon the superclass' method as part of its own implementation.

input focus -(1) The area of a window where user interaction is possible using an input device, such as a mouse or the keyboard. (2) The position in the *active window* where a user's normal interaction with the keyboard will appear.

input router -An internal OS/2 process that removes messages from the system queue.

input/output control -A device-specific command that requests a function of a device driver.

installable file system (IFS) -A file system in which software is installed when the operating system is started.

instance -(Or object instance). A specific object, as distinguished from the abstract definition of an object referred to as its class.

instance method -A method valid for a particular object.

instruction pointer -In System/38, a pointer that provides addressability for a machine interface instruction in a program.

integer atom -An *atom* that represents a predefined system constant and carries no storage overhead. For example, names of window classes provided by Presentation Manager are expressed as integer atoms.

interactive graphics -Graphics that can be moved or manipulated by a user at a terminal.

interactive program -(1) A program that is running (active) and is ready to receive (or is receiving) input from a user. (2) A running program that can receive input from the keyboard or another input device. Compare with *active program* and contrast with *noninteractive* program.

Also known as a *foreground program* .

interchange file -A file containing data that can be sent from one Presentation Manager interface application to another.

Interface Definition Language (IDL) -Language-neutral interface specification for a SOM class.

interpreter -A program that translates and executes each instruction of a high-level programming language before it translates and executes.

interprocess communication (IPC) -In the OS/2 operating system, the exchange of information between processes or threads through semaphores, pipes, queues, and shared memory.

interval timer -(1) A timer that provides program interruptions on a program-controlled basis. (2) An electronic counter that counts intervals of time under program control.

IOCtl -Input/output control.

IOPL -Input/output privilege level.

IOPL code segment -An IOPL executable section of programming code that enables an application to directly manipulate hardware interrupts and ports without replacing the device driver. See also *privilege level* .

IPC -Interprocess communication.

IPF -Information Presentation Facility.

IPF compiler -A text compiler that interpret tags in a source file and converts the information into the specified format.

IPF tag language -A markup language that provides the instructions for displaying online information.

item -A data object that can be passed in a DDE transaction.

Glossary - J

journal -A special-purpose file that is used to record changes made in the system.

Glossary - K

Kanji -A graphic character set used in Japanese ideographic alphabets.

KBD$ -Character-device name reserved for the keyboard.

kernel -The part of an operating system that performs basic functions, such as allocating hardware resources.

kerning -The design of graphics characters so that their character boxes overlap. Used to space text proportionally.

keyboard accelerator -A keystroke that generates a command message for an application.

keyboard augmentation -A function that enables a user to press a keyboard key while pressing a mouse button.

keyboard focus -A temporary attribute of a window. The window that has a keyboard focus receives all keyboard input until the focus changes to a different window.

Keys help -In SAA Common User Access architecture, a help action that provides a listing of the application keys and their assigned functions.

Glossary - L

label -In a graphics segment, an identifier of one or more elements that is used when editing the segment.

LAN -Local area network.

language support procedure -A function provided by the Presentation Manager Interface for applications that do not, or cannot (as in the case of COBOL and FORTRAN programs), provide their own dialog or window procedures.

lazy drag -See *pickup and drop* .

lazy drag set -See *pickup set* .

LDT -In the OS/2 operating system, Local Descriptor Table.

LIFO stack -A stack from which data is retrieved in last-in, first-out order.

linear address -A unique value that identifies the memory object.

linked list -Synonym for *chained list* .

list box -In SAA Advanced Common User Access architecture, a control that contains scrollable choices from which a user can select one choice.

Note: In CUA architecture, this is a programmer term. The end user term is selection list.

list button -A button labeled with an underlined down-arrow that presents a list of valid objects or choices that can be selected for that field.

list panel -A defined panel type that displays a list of items from which users can select one or more choices and then specify one or more actions to work on those choices.

load time -The point in time at which a program module is loaded into main storage for execution.

load-on-call -A function of a linkage editor that allows selected segments of the module to be disk resident while other segments are executing. Disk resident segments are loaded for execution and given control when any entry point that they contain is called.

local area network (LAN) -(1) A computer network located on a user's premises within a limited geographical area. Communication within a local area network is not subject to external regulations; however, communication across the LAN boundary may be subject to some form of regulation. (T)

Note: A LAN does not use store and forward techniques. (2) A network in which a set of devices are connected to one another for communication and that can be connected to a larger network.

Local Descriptor Table (LDT) -Defines code and data segments specific to a single task.

lock -A serialization mechanism by means of which a resource is restricted for use by the holder of the lock.

logical storage device -A device that the user can map to a physical (actual) device.

LPT1, LPT2, LPT3 -Character-device names reserved for parallel printers 1 through 3.

Glossary - M

main window -The window that is positioned relative to the *desktop window* .

manipulation button -The button on a pointing device a user presses to directly manipulate an object.

map -(1) A set of values having a defined correspondence with the quantities or values of another set. (I) (A) (2) To establish a set of values having a defined correspondence with the quantities or values of another set. (I)

marker box -In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single marker from a marker set.

marker symbol -A symbol centered on a point. Graphs and charts can use marker symbols to indicate the plotted points.

marquee box -The rectangle that appears during a selection technique in which a user selects objects by drawing a box around them with a pointing device.

Master Help Index -In the OS/2 operating system, an alphabetic list of help topics related to using the operating system.

maximize -To enlarge a window to its largest possible size.

media window -The part of the physical device (display, printer, or plotter) on which a picture is presented.

memory block -Part memory within a heap.

memory device context -A logical description of a data destination that is a memory bit map. See also *device context* .

memory management -A feature of the operating system for allocating, sharing, and freeing main storage.

memory object -Logical unit of memory requested by an application, which forms the granular unit of memory manipulation from the application viewpoint.

menu -In SAA Advanced Common User Access architecture, an extension of the menu bar that displays a list of choices available for a selected choice in the menu bar. After a user selects a choice in menu bar, the corresponding menu appears. Additional pop-up windows can appear from menu choices.

menu bar -In SAA Advanced Common User Access architecture, the area near the top of a window, below the title bar and above the rest of the window, that contains choices that provide access to other menus.

menu button -The button on a pointing device that a user presses to view a pop-up menu associated with an object.

message -(1) In the Presentation Manager, a packet of data used for communication between the Presentation Manager interface and Presentation Manager applications (2) In a user interface, information not requested by users but presented to users by the computer in response to a user action or internal process.

message box -(1) A dialog window predefined by the system and used as a simple interface for applications, without the necessity of creating dialog-template resources or dialog procedures. (2) (D of C) In SAA Advanced Common User Access architecture, a type of window that shows messages to users. See also *dialog box, primary window, secondary window* .

message filter -The means of selecting which messages from a specific window will be handled by the application.

message queue -A sequenced collection of messages to be read by the application.

message stream mode -A method of operation in which data is treated as a stream of messages. Contrast with *byte stream* .

metacharacter -See *global file-name character* .

metaclass -A class whose instances are all classes. In SOM, any class descended from SOMClass is a metaclass. The methods of a metaclass are sometimes called "class" methods.

metafile -A file containing a series of attributes that set color, shape and size, usually of a picture or a drawing. Using a program that can interpret these attributes, a user can view the assembled image.

metafile device context -A logical description of a data destination that is a metafile, which is used for graphics interchange. See also *device context* .

metalanguage -A language used to specify another language. For example, data types can be described using a metalanguage so as to make the descriptions independent of any one computer language.

method -One of the units that makes up the behavior of an object. A method is a combination of a function and a name, such that many different functions can have the same name. Which function the name refers to at any point in time depends on the object that is to execute the method and is the subject of method resolution.

method override -The replacement, by a child class, of the implementation of a method inherited from a parent and an ancestor class.

mickey -A unit of measurement for physical mouse motion whose value depends on the mouse device driver currently loaded.

micro presentation space -A graphics presentation space in which a restricted set of the GPI function calls is available.

minimize -To remove from the screen all windows associated with an application and replace them with an icon that represents the application.

mix -An attribute that determines how the foreground of a graphic primitive is combined with the existing color of graphics output. Also known as *foreground mix* . Contrast with *background mix* .

mixed character string -A string containing a mixture of one-byte and *Kanji* or Hangeul (two-byte) characters.

mnemonic -(1) A method of selecting an item on a pull-down by means of typing the highlighted letter in the menu item. (2) (D of C) In SAA Advanced Common User Access architecture, usually a single character, within the text of a choice, identified by an underscore beneath the character. If all characters in a choice already serve as mnemonics for other choices, another character, placed in parentheses immediately following the choice, can be used. When a user types the mnemonic for a choice, the choice is either selected or the cursor is moved to that choice.

modal dialog box -In SAA Advanced Common User Access architecture, a type of movable window, fixed in size, that requires a user to enter information before continuing to work in the application window from which it was displayed. Contrast with *modeless dialog box* . Also known as a *serial dialog box* . Contrast with *parallel dialog box* .

Note: In CUA architecture, this is a programmer term. The end user term is pop-up window.

model space -See *graphics model space* .

modeless dialog box -In SAA Advanced Common User Access architecture, a type of movable window, fixed in size, that allows users to continue their dialog with the application without entering information in the dialog box. Also known as a *parallel dialog box* . Contrast with *modal dialog box* .

Note: In CUA architecture, this is a programmer term. The end user term is pop-up window.

module definition file -A file that describes the code segments within a load module. For example, it indicates whether a code segment is loadable before module execution begins (preload), or loadable only when referred to at run time (load-on-call).

mouse -In SAA usage, a device that a user moves on a flat surface to position a pointer on the screen. It allows a user to select a choice o function to be performed or to perform operations on the screen, such as dragging or drawing lines from one position to another.

MOUSE$ -Character-device name reserved for a mouse.

multiple-choice selection -In SAA Basic Common User Access architecture, a type of field from which a user can select one or more choices or select none. See also *check box* . Contrast with *extended-choice selection* .

multiple-line entry field -In SAA Advanced Common User Access architecture, a control into which a user types more than one line of information. See also *single-line entry field* .

multitasking -The concurrent processing of applications or parts of applications. A running application and its data are protected from other concurrently running applications.

mutex semaphore -(Mutual exclusion semaphore). A semaphore that enables threads to serialize their access to resources. Only the thread that currently owns the mutex semaphore can gain access to the resource, thus preventing one thread from interrupting operations being performed by another.

muxwait semaphore -(Multiple wait semaphore). A semaphore that enables a thread to wait either for multiple event semaphores to be posted or for multiple mutex semaphores to be released. Alternatively, a muxwait semaphore can be set to enable a thread to wait for any ONE of the event or mutex semaphores in the muxwait semaphore's list to be posted or released.

Glossary - N

named pipe -A named buffer that provides client-to-server, server-to-client, or full duplex communication between unrelated processes. Contrast with *unnamed pipe* .

national language support (NLS) -The modification or conversion of a United States English product to conform to the requirements of another language or country. This can include the enabling or retrofitting of a product and the translation of nomenclature, MRI, or documentation of a product.

nested list -A list that is contained within another list.

NLS -national language support.

non-8.3 file-name format -A file-naming convention in which file names can consist of up to 255 characters. See also *8.3 file-name format* .

noncritical extended attribute -An extended attribute that is not necessary for the function of an application.

nondestructive read -Reading that does not erase the data in the source location. (T)

noninteractive program -A running program that cannot receive input from the keyboard or other input device. Compare with *active* program, and contrast with *interactive program* .

nonretained graphics -Graphic primitives that are not remembered by the Presentation Manager interface when they have been drawn. Contrast with *retained graphics* .

null character (NUL) -(1) Character-device name reserved for a nonexistent (dummy) device. (2) (D of C) A control character that is used to accomplish media-fill or time-fill and that may be inserted into or removed from a sequence of characters without affecting the meaning of the sequence; however, the control of equipment or the format may be affected by this character. (I) (A)

null-terminated string -A string of (n+1) characters where the (n+1)th character is the 'null' character (0x00) Also known as 'zero-terminated' string and 'ASCIIZ' string.

Glossary - O

object -The elements of data and function that programs create, manipulate, pass as arguments, and so forth. An object is a way of associating specific data values with a specific set of named functions (called *methods*) for a period of time (referred to as the *lifetime* of the object). The data values of an object are referred to as its *state* . In SOM, objects are created by other objects called *classes* . The

specification of what comprises the set of functions and data elements that make up an object is referred to as the *definition* of a class.

SOM objects offer a high degree of *encapsulation*. This property permits many aspects of the implementation of an object to change without affecting client programs that depend on the object's behavior.

object definition -See *class*.

object instance -See *instance*.

Object Interface Definition Language (OIDL) -Specification language used in SOM Version 1 for defining classes. Replaced by Interface Definition Language (IDL).

object window -A window that does not have a parent but which might have child windows. An object window cannot be presented on a device.

OIDL -Object Interface Definition Language.

open -To start working with a file, directory, or other object.

ordered list -Vertical arrangements of items, with each item in the list preceded by a number or letter.

outline font -A set of symbols, each of which is created as a series of lines and curves. Synonymous with *vector font*. Contrast with *image font*.

output area -An area of storage reserved for output. (A)

owner window -A window into which specific events that occur in another (owned) window are reported.

ownership -The determination of how windows communicate using messages.

owning process -The process that owns the resources that might be shared with other processes.

Glossary - P

page -(1) A 4KB segment of contiguous physical memory. (2) (D of C) A defined unit of space on a storage medium.

page viewport -A boundary in device coordinates that defines the area of the output device in which graphics are to be displayed. The presentation-page contents are transformed automatically to the page viewport in device space.

paint -(1) The action of drawing or redrawing the contents of a window. (2) In computer graphics, to shade an area of a display image; for example, with crosshatching or color.

panel -In SAA Basic Common User Access architecture, a particular arrangement of information that is presented in a window or pop-up. If some of the information is not visible, a user can scroll through the information.

panel area -An area within a panel that contains related information. The three major Common User Access-defined panel areas are the action bar, the function key area, and the panel body.

panel area separator -In SAA Basic Common User Access architecture, a solid, dashed, or blank line that provides a visual distinction between two adjacent areas of a panel.

panel body -The portion of a panel not occupied by the action bar, function key area, title or scroll bars. The panel body can contain protected information, selection fields, and entry fields. The layout and content of the panel body determine the panel type.

panel body area -See *client area*.

panel definition -A description of the contents and characteristics of a panel. A panel definition is the application developer's mechanism for predefining the format to be presented to users in a window.

panel ID -In SAA Basic Common User Access architecture, a panel identifier, located in the upper-left corner of a panel. A user can choose whether to display the panel ID.

panel title -In SAA Basic Common User Access architecture, a particular arrangement of information that is presented in a window or pop-up. If some of the information is not visible, a user can scroll through the information.

paper size -The size of paper, defined in either standard U.S. or European names (for example, A, B, A4), and measured in inches or millimeters respectively.

parallel dialog box -See *modeless dialog box*.

parameter list -A list of values that provides a means of associating addressability of data defined in a called program with data in the calling program. It contains parameter names and the order in which they are to be associated in the calling and called program.

parent class -See *inheritance* .

parent process -In the OS/2 operating system, a process that creates other processes. Contrast with *child process* .

parent window -In the OS/2 operating system, a window that creates a child window. The child window is drawn within the parent window. If the parent window is moved, resized, or destroyed, the child window also will be moved, resized, or destroyed. However, the child window can be moved and resized independently from the parent window, within the boundaries of the parent window. Contrast with *child window* .

partition -(1) A fixed-size division of storage. (2) On an IBM personal computer fixed disk, one of four possible storage areas of variable size; one may be accessed by DOS, and each of the others may be assigned to another operating system.

Paste -A choice in the Edit pull-down that a user selects to move the contents of the clipboard into a preselected location. See also *Copy* and *Cut* .

path -The route used to locate files; the storage location of a file. A fully qualified path lists the drive identifier, directory name, subdirectory name (if any), and file name with the associated extension.

PDD -Physical device driver.

peeking -An action taken by any thread in the process that owns the queue to examine queue elements without removing them.

pel -(1) The smallest area of a display screen capable of being addressed and switched between visible and invisible states. Synonym for *display point* , *pixel* , and *picture element* . (2) (D of C) Picture element.

persistent object -An object whose instance data and state are preserved between system shutdown and system startup.

physical device driver (PDD) -A system interface that handles hardware interrupts and supports a set of input and output functions.

pick -To select part of a displayed object using the pointer.

pickup -To add an object or set of objects to the pickup set.

pickup and drop -A drag operation that does not require the direct manipulation button to be pressed for the duration of the drag.

pickup set -The set of objects that have been picked up as part of a pickup and drop operation.

picture chain -See *segment chain* .

picture element -(1) Synonym for *pel* . (2) (D of C) In computer graphics, the smallest element of a display surface that can be independently assigned color and intensity. (T) . (3) The area of the finest detail that can be reproduced effectively on the recording medium.

PID -Process identification.

pipe -(1) A named or unnamed buffer used to pass data between processes. A process reads from or writes to a pipe as if the pipe were a standard-input or standard-output file. See also *named pipe* and *unnamed pipe* . (2) (D of C) To direct data so that the output from one process becomes the input to another process. The standard output of one command can be connected to the standard input of another with the pipe operator (|).

pixel -(1) Synonym for *pel* . (2) (D of C) Picture element.

plotter -An output unit that directly produces a hardcopy record of data on a removable medium, in the form of a two-dimensional graphic representation. (T)

PM -Presentation Manager.

pointer -(1) The symbol displayed on the screen that is moved by a pointing device, such as a *mouse* . The pointer is used to point at items that users can select. Contrast with *cursor* . (2) A data element that indicates the location of another data element. (T)

POINTER$ -Character-device name reserved for a pointer device (mouse screen support).

pointing device -In SAA Advanced Common User Access architecture, an instrument, such as a mouse, trackball, or joystick, used to move a pointer on the screen.

pointings -Pairs of x-y coordinates produced by an operator defining positions on a screen with a pointing device, such as a *mouse* .

polyfillet -A curve based on a sequence of lines. The curve is tangential to the end points of the first and last lines, and tangential also to the midpoints of all other lines. See also *fillet* .

polygon -One or more closed figures that can be drawn filled, outlined, or filled and outlined.

polyline -A sequence of adjoining lines.

polymorphism -The ability to have different implementations of the same method for two or more classes of objects.

pop -To retrieve an item from a last-in-first-out stack of items. Contrast with *push* .

pop-up menu -A menu that lists the actions that a user can perform on an object. The contents of the pop-up menu can vary depending on the context, or state, of the object.

pop-up window -(1) A window that appears on top of another window in a dialog. Each pop-up window must be completed before returning to the underlying window. (2) (D of C) In SAA Advanced Common User Access architecture, a movable window, fixed in size, in which a user provides information required by an application so that it can continue to process a user request.

presentation drivers -Special purpose I/O routines that handle field device-independent I/O requests from the PM and its applications.

Presentation Manager (PM) -The interface of the OS/2 operating system that presents, in windows a graphics-based interface to applications and files installed and running under the OS/2 operating system.

presentation page -The coordinate space in which a picture is assembled for display.

presentation space (PS) -(1) Contains the device-independent definition of a picture. (2) (D of C) The display space on a display device.

primary window -In SAA Common User Access architecture, the window in which the main interaction between the user and the application takes place. In a multiprogramming environment, each application starts in its own primary window. The primary window remains for the duration of the application, although the panel displayed will change as the user's dialog moves forward. See also *secondary window* .

primitive -In computer graphics, one of several simple functions for drawing on the screen, including, for example, the rectangle, line, ellipse, polygon, and so on.

primitive attribute -A specifiable characteristic of a graphic primitive. See *graphics attributes* .

print job -The result of sending a document or picture to be printed.

Print Manager -In the Presentation Manager, the part of the spooler that manages the spooling process. It also allows users to view print queues and to manipulate print jobs.

privilege level -A protection level imposed by the hardware architecture of the IBM personal computer. There are four privilege levels (number 0 through 3). Only certain types of programs are allowed to execute at each privilege level. See also *IOPL code segment* .

procedure call -In programming languages, a language construct for invoking execution of a procedure.

process -An instance of an executing application and the resources it is using.

program -A sequence of instructions that a computer can interpret and execute.

program details -Information about a program that is specified in the *Program Manager* window and is used when the program is started.

program group -In the Presentation Manager, several programs that can be acted upon as a single entity.

program name -The full file specification of a program. Contrast with *program title* .

program title -The name of a program as it is listed in the *Program Manager* window. Contrast with *program name* .

prompt -A displayed symbol or message that requests input from the user or gives operational information; for example, on the display screen of an IBM personal computer, the DOS A> prompt. The user must respond to the prompt in order to proceed.

protect mode -A method of program operation that limits or prevents access to certain instructions or areas of storage. Contrast with *real mode* .

protocol -A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I)

pseudocode -An artificial language used to describe computer program algorithms without using the syntax of any particular programming language. (A)

pull-down -(1) An *action bar* extension that displays a list of choices available for a selected action bar choice. After users select an action bar choice, the pull-down appears with the list of choices. Additional *pop-up windows* may appear from pull-down choices to further extend the actions available to users. (2) (D of C) In SAA Common User Access architecture, pertaining to a choice in an action bar pull-down.

push -To add an item to a last-in-first-out stack of items. Contrast with *pop* .

push button -In SAA Advanced Common User Access architecture, a rectangle with text inside. Push buttons are used in windows for actions that occur immediately when the push button is selected.

putback -To remove an object or set of objects from the lazy drag set. This has the effect of undoing the pickup operation for those objects

putdown -To drop the objects in the lazy drag set on the target object.

Glossary - Q

queue -(1) A linked list of elements waiting to be processed in FIFO order. For example, a queue may be a list of print jobs waiting to be printed. (2) (D of C) A line or list of items waiting to be processed; for example, work to be performed or messages to be displayed.

queued device context -A logical description of a data destination (for example, a printer or plotter) where the output is to go through the spooler. See also *device context* .

Glossary - R

radio button -(1) A control window, shaped like a round button on the screen, that can be in a checked or unchecked state. It is used to select a single item from a list. Contrast with *check box* . (2) In SAA Advanced Common User Access architecture, a circle with text beside it. Radio buttons are combined to show a user a fixed set of choices from which only one can be selected. The circle is partially filled when a choice is selected.

RAS -Reliability, availability, and serviceability.

raster -(1) In computer graphics, a predetermined pattern of lines that provides uniform coverage of a display space. (T) (2) The coordinate grid that divides the display area of a display device. (A)

read-only file -A file that can be read from but not written to.

real mode -A method of program operation that does not limit or prevent access to any instructions or areas of storage. The operating system loads the entire program into storage and gives the program access to all system resources. Contrast with *protect mode* .

realize -To cause the system to ensure, wherever possible, that the physical color table of a device is set to the closest possible match in the logical color table.

recursive routine -A routine that can call itself, or be called by another routine that was called by the recursive routine.

reentrant -The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

reference phrase -(1) A word or phrase that is emphasized in a device-dependent manner to inform the user that additional information for the word or phrase is available. (2) (D of C) In hypertext, text that is highlighted and preceded by a single-character input field used to signify the existence of a hypertext link.

reference phrase help -In SAA Common User Access architecture, highlighted words or phrases within help information that a user selects to get additional information.

refresh -To update a window, with changed information, to its current status.

region -A clipping boundary in device space.

register -A part of internal storage having a specified storage capacity and usually intended for a specific purpose. (T)

remote file system -A file-system driver that gains access to a remote system without a block device driver.

resource -The means of providing extra information used in the definition of a window. A resource can contain definitions of fonts, templates, accelerators, and mnemonics; the definitions are held in a resource file.

resource file -A file containing information used in the definition of a window. Definitions can be of fonts, templates, accelerators, and mnemonics.

restore -To return a window to its original size or position following a sizing or moving action.

retained graphics -Graphic primitives that are remembered by the Presentation Manager interface after they have been drawn. Contrast with *nonretained graphics* .

return code -(1) A value returned to a program to indicate the results of an operation requested by that program. (2) A code used to influence the execution of succeeding instructions.(A)

reverse video -(1) A form of highlighting a character, field, or cursor by reversing the color of the character, field, or cursor with its background; for example, changing a red character on a black background to a black character on a red background. (2) In SAA Basic Common User Access architecture, a screen emphasis feature that interchanges the foreground and background colors of an item.

REXX Language -Restructured Extended Executor. A procedural language that provides batch language functions along with structured programming constructs such as loops; conditional testing and subroutines.

RGB -(1) Color coding in which the brightness of the additive primary colors of light, red, green, and blue, are specified as three distinct values of white light. (2) Pertaining to a color display that accepts signals representing red, green, and blue.

roman -Relating to a type style with upright characters.

root segment -In a hierarchical database, the highest segment in the tree structure.

round-robin scheduling -A process that allows each thread to run for a specified amount of time.

run time -(1) Any instant at which the execution of a particular computer program takes place. (T) (2) The amount of time needed for the execution of a particular computer program. (T) (3) The time during which an instruction in an instruction register is decoded and performed. Synonym for *execution time* .

Glossary - S

SAA -Systems Application Architecture.

SBCS -Single-byte character set.

scheduler -A computer program designed to perform functions such as scheduling, initiation, and termination of jobs.

screen -In SAA Basic Common User Access architecture, the physical surface of a display device upon which information is shown to a user.

screen device context -A logical description of a data destination that is a particular window on the screen. See also *device context* .

SCREEN$ -Character-device name reserved for the display screen.

scroll bar -In SAA Advanced Common User Access architecture, a part of a window, associated with a scrollable area, that a user interacts with to see information that is not currently allows visible.

scrollable entry field -An entry field larger than the visible field.

scrollable selection field -A selection field that contains more choices than are visible.

scrolling -Moving a display image vertically or horizontally in a manner such that new data appears at one edge, as existing data disappears at the opposite edge.

secondary window -A window that contains information that is dependent on information in a primary window and is used to supplement the interaction in the primary window.

sector -On disk or diskette storage, an addressable subdivision of a track used to record one block of a program or data.

segment -See *graphics segment* .

segment attributes -Attributes that apply to the segment as an entity, as opposed to the individual primitives within the segment. For example, the visibility or detectability of a segment.

segment chain -All segments in a graphics presentation space that are defined with the 'chained' attribute. Synonym for *picture chain* .

segment priority -The order in which segments are drawn.

segment store -An area in a normal graphics presentation space where retained graphics segments are stored.

select -To mark or choose an item. Note that *select* means to mark or type in a choice on the screen; *enter* means to send all selected choices to the computer for processing.

select button -The button on a pointing device, such as a mouse, that is pressed to select a menu choice. Also known as button 1.

selection cursor -In SAA Advanced Common User Access architecture, a visual indication that a user has selected a choice. It is represented by outlining the choice with a dotted box. See also *text cursor* .

selection field -(1) In SAA Advanced Common User Access architecture, a set of related choices. See also *entry field*. (2) In SAA Basic Common User Access architecture, an area of a panel that cannot be scrolled and contains a fixed number of choices.

semantics -The relationships between symbols and their meanings.

semaphore -An object used by applications for signalling purposes and for controlling access to serially reusable resources.

separator -In SAA Advanced Common User Access architecture, a line or color boundary that provides a visual distinction between two adjacent areas.

serial dialog box -See *modal dialog box*.

serialization -The consecutive ordering of items.

serialize -To ensure that one or more events occur in a specified sequence.

serially reusable resource (SRR) -A logical resource or object that can be accessed by only one task at a time.

session -(1) A routing mechanism for user interaction via the console; a complete environment that determines how an application runs and how users interact with the application. OS/2 can manage more than one session at a time, and more than one process can run in a session. Each session has its own set of environment variables that determine where OS/2 looks for dynamic-link libraries and other important files. (2) (D of C) In the OS/2 operating system, one instance of a started program or command prompt. Each session is separate from all other sessions that might be running on the computer. The operating system is responsible for coordinating the resources that each session uses, such as computer memory, allocation of processor time, and windows on the screen.

Settings Notebook -A control window that is used to display the settings for an object and to enable the user to change them.

shadow -An object that refers to another object. A shadow is not a copy of another object, but is another representation of the object.

shadow box -The area on the screen that follows mouse movements and shows what shape the window will take if the mouse button is released.

shared data -Data that is used by two or more programs.

shared memory -In the OS/2 operating system, a segment that can be used by more than one program.

shear -In computer graphics, the forward or backward slant of a graphics symbol or string of such symbols relative to a line perpendicular to the baseline of the symbol.

shell -(1) A software interface between a user and the operating system of a computer. Shell programs interpret commands and user interactions on devices such as keyboards, pointing devices, and touch-sensitive screens, and communicate them to the operating system. (2) Software that allows a kernel program to run under different operating-system environments.

shutdown -The process of ending operation of a system or a subsystem, following a defined procedure.

sibling processes -Child processes that have the same parent process.

sibling windows -Child windows that have the same parent window.

simple list -A list of like values; for example, a list of user names. Contrast with *mixed list*.

single-byte character set (SBCS) -A character set in which each character is represented by a one-byte code. Contrast with *double-byte character set*.

slider box -In SAA Advanced Common User Access architecture: a part of the scroll bar that shows the position and size of the visible information in a window relative to the total amount of information available. Also known as *thumb mark*.

SOM -System Object Model.

source file -A file that contains source statements for items such as high-level language programs and data description specifications.

source statement -A statement written in a programming language.

specific dynamic-link module -A dynamic-link module created for the exclusive use of an application.

spin button -In SAA Advanced Common User Access architecture, a type of entry field that shows a scrollable ring of choices from which a user can select a choice. After the last choice is displayed, the first choice is displayed again. A user can also type a choice from the scrollable ring into the entry field without interacting with the spin button.

spline -A sequence of one or more Bézier curves.

spooler -A program that intercepts the data going to printer devices and writes it to disk. The data is printed or plotted when it is complete and the required device is available. The spooler prevents output from different sources from being intermixed.

stack -A list constructed and maintained so that the next data element to be retrieved is the most recently stored. This method is characterized as last-in-first-out (LIFO).

standard window -A collection of window elements that form a panel. The standard window can include one or more of the following window elements: sizing borders, system menu icon, title bar, maximize/minimize/restore icons, action bar and pull-downs, scroll bars, and client area.

static control -The means by which the application presents descriptive information (for example, headings and descriptors) to the user. The user cannot change this information.

static storage -(1) A read/write storage unit in which data is retained in the absence of control signals. (A) Static storage may use dynamic addressing or sensing circuits. (2) Storage other than *dynamic storage* . (A)

style -See *window style* .

subclass -A class that inherits from another class. See also *Inheritance* .

subdirectory -In an IBM personal computer, a file referred to in a root directory that contains the names of other files stored on the diskette or fixed disk.

superclass -A class from which another class inherits. See also *inheritance* .

swapping -(1) A process that interchanges the contents of an area of real storage with the contents of an area in auxiliary storage. (I) (A) (2) In a system with virtual storage, a paging technique that writes the active pages of a job to auxiliary storage and reads pages of another job from auxiliary storage into real storage. (3) The process of temporarily removing an active job from main storage, saving it on disk, and processing another job in the area of main storage formerly occupied by the first job.

switch -(1) In SAA usage, to move the cursor from one point of interest to another; for example, to move from one screen or window to another or from a place within a displayed image to another place on the same displayed image. (2) In a computer program, a conditional instruction and an indicator to be interrogated by that instruction. (3) A device or programming technique for making a selection, for example, a toggle, a conditional jump.

switch list -See *Task List* .

symbolic identifier -A text string that equates to an integer value in an include file, which is used to identify a programming object.

symbols -In Information Presentation Facility, a document element used to produce characters that cannot be entered from the keyboard.

synchronous -Pertaining to two or more processes that depend upon the occurrence of specific events such as common timing signals. (T) See also *asynchronous* .

System Menu -In the Presentation Manager, the pull-down in the top left corner of a window that allows it to be moved and sized with the keyboard.

System Object Model (SOM) -A mechanism for language-neutral, object-oriented programming in the OS/2 environment.

system queue -The master queue for all pointer device or keyboard events.

system-defined messages -Messages that control the operations of applications and provides input an other information for applications to process.

Systems Application Architecture (SAA) -A set of IBM software interfaces, conventions, and protocols that provide a framework for designing and developing applications that are consistent across systems.

Glossary - T

table tags -In Information Presentation Facility, a document element that formats text in an arrangement of rows and columns.

tag -(1) One or more characters attached to a set of data that contain information about the set, including its identification. (I) (A) (2) In Generalized Markup Language markup, a name for a type of document or document element that is entered in the source document to identify it.

target object -An object to which the user is transferring information.

Task List -In the Presentation Manager, the list of programs that are active. The list can be used to switch to a program and to stop programs.

terminate-and-stay-resident (TSR) -Pertaining to an application that modifies an operating system interrupt vector to point to its own location (known as hooking an interrupt).

text -Characters or symbols.

text cursor -A symbol displayed in an entry field that indicates where typed input will appear.

text window -Also known as the VIO window.

text-windowed application -The environment in which the operating system performs advanced-video input and output operations.

thread -A unit of execution within a process. It uses the resources of the process.

thumb mark -The portion of the scroll bar that describes the range and properties of the data that is currently visible in a window. Also known as a *slider box* .

thunk -Term used to describe the process of address conversion, stack and structure realignment, etc., necessary when passing control between 16-bit and 32-bit modules.

tilde -A mark used to denote the character that is to be used as a mnemonic when selecting text items within a menu.

time slice -(1) An interval of time on the processing unit allocated for use in performing a task. After the interval has expired, processing-unit time is allocated to another task, so a task cannot monopolize processing-unit time beyond a fixed limit. (2) In systems with time sharing, a segment of time allocated to a terminal job.

time-critical process -A process that must be performed within a specified time after an event has occurred.

timer -A facility provided under the Presentation Manager, whereby Presentation Manager will dispatch a message of class WM_TIMER to a particular window at specified intervals. This capability may be used by an application to perform a specific processing task at predetermined intervals, without the necessity for the application to explicitly keep track of the passage of time.

timer tick -See *clock tick* .

title bar -In SAA Advanced Common User Access architecture, the area at the top of each window that contains the window title and system menu icon. When appropriate, it also contains the minimize, maximize, and restore icons. Contrast with *panel title* .

TLB -Translation lookaside buffer.

transaction -An exchange between a workstation and another device that accomplishes a particular action or result.

transform -(1) The action of modifying a picture by scaling, shearing, reflecting, rotating, or translating. (2) The object that performs or defines such a modification; also referred to as a *transformation* .

Translation lookaside buffer (TLB) -A hardware-based address caching mechanism for paging information.

Tree -In the Presentation Manager, the window in the *File Manager* that shows the organization of drives and directories.

truncate -(1) To terminate a computational process in accordance with some rule (A) (2) To remove the beginning or ending elements of a string. (3) To drop data that cannot be printed or displayed in the line width specified or available. (4) To shorten a field or statement to a specified length.

TSR -Terminate-and-stay-resident.

Glossary - U

unnamed pipe -A circular buffer, created in memory, used by related processes to communicate with one another. Contrast with *named* pipe.

unordered list -In Information Presentation Facility, a vertical arrangement of items in a list, with each item in the list preceded by a special character or bullet.

update region -A system-provided area of dynamic storage containing one or more (not necessarily contiguous) rectangular areas of a window that are visually invalid or incorrect, and therefore are in need of repainting.

user interface -Hardware, software, or both that allows a user to interact with and perform operations on a system, program, or device.

User Shell -A component of OS/2 that uses a graphics-based, windowed interface to allow the user to manage applications and files installed and running under OS/2.

utility program -(1) A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program. (T) (2) A program designed to perform an everyday task such as copying data from one storage device to another. (A)

Glossary - V

value set control -A visual component that enables a user to select one choice from a group of mutually exclusive choices.

vector font -A set of symbols, each of which is created as a series of lines and curves. Synonymous with *outline font* . Contrast with *image font* .

VGA -Video graphics array.

view -A way of looking at an object's information.

viewing pipeline -The series of transformations applied to a graphic object to map the object to the device on which it is to be presented.

viewing window -A clipping boundary that defines the visible part of model space.

VIO -Video Input/Output.

virtual memory (VM) -Synonymous with *virtual storage* .

virtual storage -(1) The storage space that may be regarded as addressable main storage by the user of a computer system in which virtual addresses are mapped into real addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of auxiliary storage available, not by the actual number of main storage locations. (I) (A) (2) Addressable space that is apparent to the user as the processor storage space, from which the instructions and the data are mapped into the processor storage locations. (3) Synonymous with *virtual memory* .

visible region -A window's presentation space, clipped to the boundary of the window and the boundaries of any overlying window.

volume -(1) A file-system driver that uses a block device driver for input and output operations to a local or remote device. (I) (2) A portion of data, together with its data carrier, that can be handled conveniently as a unit.

Glossary - W

wildcard character -Synonymous with *global file-name character* .

window -(1) A portion of a display surface in which display images pertaining to a particular application can be presented. Different applications can be displayed simultaneously in different windows. (A) (2) An area of the screen with visible boundaries within which information is displayed. A window can be smaller than or the same size as the screen. Windows can appear to overlap on the screen.

window class -The grouping of windows whose processing needs conform to the services provided by one window procedure.

window coordinates -A set of coordinates by which a window position or size is defined; measured in device units, or *pels* .

window handle -Unique identifier of a window, generated by Presentation Manager when the window is created, and used by applications to direct messages to the window.

window procedure -Code that is activated in response to a message. The procedure controls the appearance and behavior of its associated windows.

window rectangle -The means by which the size and position of a window is described in relation to the desktop window.

window resource -A read-only data segment stored in the .EXE file of an application o the .DLL file of a dynamic link library.

window style -The set of properties that influence how events related to a particular window will be processed.

window title -In SAA Advanced Common User Access architecture, the area in the title bar that contains the name of the application and the OS/2 operating system file name, if applicable.

Workplace Shell -The OS/2 object-oriented, graphical user interface.

workstation -(1) A display screen together with attachments such as a keyboard, a local copy device, or a tablet. (2) (D of C) One or more programmable or nonprogrammable devices that allow a user to do work.

world coordinates -A device-independent Cartesian coordinate system used by the application program for specifying graphical input and output. (I) (A)

world-coordinate space -Coordinate space in which graphics are defined before transformations are applied.

WYSIWYG -What-You-See-Is-What-You-Get. A capability of a text editor to continually display pages exactly as they will be printed.

Glossary - X

There are no glossary terms for this starting letter.

Glossary - Y

There are no glossary terms for this starting letter.

Glossary - Z

z-order -The order in which sibling windows are presented. The topmost sibling window obscures any portion of the siblings that it overlaps; the same effect occurs down through the order of lower sibling windows.

zooming -The progressive scaling of an entire display image in order to give the visual impression of movement of all or part of a display group toward or away from an observer. (I) (A)

8.3 file-name format -A file-naming convention in which file names are limited to eight characters before and three characters after a single dot. Usually pronounced "eight-dot-three." See also *non-8.3 file-name format* .
